

**Fourth International Conference on
Autonomous Agents and Multi-Agent Systems
(AAMAS)**

**7th International Workshop on
Agent-Oriented Information Systems
(AOIS)**

Utrecht, The Netherlands

26th July 2005

Brian Henderson-Sellers and Michael Winikoff (editors)

<http://www.aois.org>

Preface

Information systems have become the backbone of all kinds of organizations today. In almost every sector – manufacturing, education, health care, government, and businesses large and small– information systems are relied upon for everyday work, communication, information gathering, and decision-making. Yet the inflexibilities in current technologies and methods have also resulted in poor performance, incompatibilities, and obstacles to change. As many organizations are reinventing themselves to meet the challenges of global competition and e-commerce, there is increasing pressure to develop and deploy new technologies that are flexible, robust, and responsive to rapid and unexpected change.

Agent concepts hold great promise for responding to the new realities of information systems. They offer higher level abstractions and mechanisms which address issues such as knowledge representation and reasoning, communication, coordination, cooperation among heterogeneous and autonomous parties, perception, commitments, goals, beliefs, intentions, etc. On the one hand, the concrete implementation of these concepts can lead to advanced functionalities, e.g., in inference-based query answering, transaction control, adaptive workflows, brokering and integration of disparate information sources, and automated communication processes. On the other hand, their rich representational capabilities allow more faithful and flexible treatments of complex organizational processes, leading to more effective requirements analysis, and architectural/detailed design. The workshop focusses on how agent concepts and techniques will contribute to meeting information systems needs today and tomorrow.

Workshop Format

To foster greater communication and interaction between the Information Systems and Agents communities, we are organizing the workshop as a bi-conference event. It is intended to be a single “logical” event with two “physical” venues. It is hoped that this arrangement will encourage greater participation from, and more exchange between, both communities.

These proceedings are for the first part of the workshop, in Utrecht on the 26th of July, as part of the fourth international conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2005); the second component being scheduled for the ER2005 conference in Klagenfurt, Austria in October 2005.

For the AOIS-2005 workshop at AAMAS, 17 paper submissions were received. These were peer-reviewed, primarily by members of the programme and steering committees, and 12 papers were accepted (for an acceptance rate of 71%).

We would like to gratefully acknowledge all the contributions to the workshop: those by the authors, the participants, and the reviewers. We believe that these accepted papers reflect the field’s state of the art very well. Furthermore, we anticipate that they constitute an excellent basis for an in-depth and fruitful exchange of thoughts and ideas on the various issues of agent-oriented information systems. We would in particular like to thank Paolo Giorgini who has co-chaired the AOIS@AAMAS workshop from 2002 to 2004.

Brian Henderson-Sellers and Michael Winikoff
(Workshop co-chairs)

Workshop Chairs

Brian Henderson-Sellers
Faculty of Information Technology
University of Technology, Sydney
Email: brian@it.uts.edu.au

Michael Winikoff
School of Computer Science & IT
RMIT University
Email: winikoff@cs.rmit.edu.au

Steering Committee

Yves Lespérance
Dept. of Computer Science
York University, Canada
Email: lesperan@cs.yorku.ca

Gerd Wagner
Dept. of Technology
Management
Eindhoven University of
Technology, The Netherlands
Email: G.Wagner@tm.tue.nl

Eric Yu
Faculty of Information Studies
University of Toronto, Canada
Email: eric.yu@utoronto.ca

Paolo Giorgini
Department of Information and Communication Technology
University of Trento, Italy
Email: paolo.giorgini@dit.unitn.it

Programme Committee

C. Bernon (University Paul Sabatier, Toulouse, France)
M. B. Blake (Georgetown University, Washington DC, USA)
P. Bresciani (ITC-irst, Italy)
J. Castro (Federal University of Pernambuco, Brazil)
L. Cernuzzi (University Católica, Paraguay)
M. Cossentino (ICAR-CNR, Palermo, Italy)
L. Cysneiros (York University, Canada)
J. Debenham (University of Technology, Sydney)
S. DeLoach (Kansas State University, USA)
F. Dignum (Utrecht University, The Netherlands)
P. Donzelli (University of Maryland, USA)
B. Espinasse (LSIS UMR CNRS, Marseilles, France)
B. H. Far (University of Calgary, Canada)
I. A. Ferguson (B2B Machines, USA)
S. Faulkner (University of Namur, Belgium)
A. Garcia (University of Lancaster, UK)
C. Ghidini (ITC-irst, Italy)
A. K. Ghose (University of Wollongong, Australia)
M.-P. Gleizes (University Paul Sabatier, Toulouse, France)
C. Gonzalez-Perez (University of Technology, Sydney, Australia)
G. Guizzardi (University of Twente, Netherlands)
I. Hawryszkiewicz (University of Technology, Sydney, Australia)
C. Iglesias (Technical University of Madrid, Spain)
M. Kolp (University catholique de Louvain, Belgium)
C. Li (University of Technology, Sydney, Australia)
C. Lucena (PUC Rio, Brazil)
Ph. Massonet (CETIC, Belgium)
H. Mouratidis (University of East London, UK)
J. Müller (Siemens, Germany)
D. E. O'Leary (University of South California, USA)
A. Omicini (Università degli Studi di Bologna, Italy)

J. Pavón (Universidad Complutense Madrid, Spain)
O. F. Rana (Cardiff University, UK)
O. Shehory (IBM Haifa Labs, Israel)
N. Szirbik (Technische Universiteit Eindhoven, The Netherlands)
V. Torres da Silva (PUC Rio, Brazil)
N. Tran (UNSW, Australia)
C. Woo (University British Columbia, Canada)
B. Yu (CMU, USA)
A. Zeid (American University of Cairo, Egypt)
Z. Zhang (Deakin University, Australia)

Additional reviewers: Renata S.S. Guizzardi, Ambra Molesini, Arnon Sturm.

Table of Contents

Distributed Storage Systems Management: An Agent Application Domain? (Invited Talk) <i>O. Shehory</i>	1
Adapted Information Retrieval in Web Information Systems using PUMAS <i>A. Carrillo Ramos, J. Gensel, M. Villanova-Oliver and H. Martin</i>	3
INCA (Investor Network Collaborative Architecture) — A Method in the Madness of Wall Street <i>S. C. Sundararajan, S. Sankarlal and A. Kumar</i>	11
An Agent-Based Meta-Level Architecture for Strategic Reasoning in Naval Planning <i>M. Hoogendoorn, C. M. Jonker, P.-P. van Maanen and J. Treur</i>	18
Design Options for Subscription Managers <i>A. Mbala, L. Padgham and M. Winikoff</i>	26
Supporting Program Indexing and Querying in Source Code Digital Libraries <i>Y. Yusof and O. F. Rana</i>	34
Architecture for Distributed Agent-Based Workflows <i>C. Reese, J. Ortmann, S. Offermann, D. Moldt, K. Lehmann and T. Carl</i>	42
OWL-P: A Methodology for Business Process Development <i>N. Desai, A. U. Mallya, A. K. Chopra and M. P. Singh</i>	50
An Ontology Support for Semantic Aware Agents <i>M. Tomaiuolo, P. Turci, F. Bergenti and A. Poggi</i>	58
On the Cost of Agent-awareness for Negotiation Services <i>A. Giovannucci and J. A. Rodríguez-Aguilar</i>	66
Automated Interpretation of Agent Behavior <i>D. N. Lam and K. S. Barber</i>	74
Requirements Analysis of an Agent's Reasoning Capability <i>T. Bosse, C. M. Jonker and J. Treur</i>	82
Identification of Reusable Method Fragments from the PASSI Agent-Oriented Methodology <i>B. Henderson-Sellers, J. Debenham, N. Tran, M. Cossentino and G. Low</i>	90

Distributed storage systems management: an agent application domain? (Invited Talk)

Onn Shehory
IBM Haifa Labs, Israel

May 16, 2005

Abstract

In recent years, the amount of data produced and stored by enterprises increases rapidly. A variety of storage technologies, systems and subsystems have been developed to address the increasing storage needs. As a result, enterprise storage systems have increased in size, complexity, and distribution. Consequently, the management of storage systems has become a complex task. Commonly, although the actual storage capacity purchased may be adequate for the storage needs of the organization, the performance of the storage system is poor. Such poor performance manifests itself in a poor quality of service, and may negatively affect business functions of the organization.

One remedy to poor system management is to increase the staffing of the system administration team. Yet, well-trained system administrators are scarce, and their cost is very high. The alternative is the use of automated management tools. However, existing storage management software tools offer a rather limited management function. Comprehensive storage resource management, including intelligent problem detection and prediction, as well as optimized re-allocation, is far from being achieved. This invites further research into the problem of storage resource management.

In this talk we will introduce the underlying concepts of enterprise distributed storage systems, the typical performance problems they introduce, and the existing solutions. We will discuss open problems and examine the relevance of agent technology to solving these problems.

Adapted Information Retrieval in Web Information Systems using PUMAS

Angela Carrillo Ramos, Jérôme Gensel, Marlène Villanova-Oliver, Hervé Martin

Laboratoire LSR - IMAG

B.P. 72 - 38402 Saint Martin d'Hères, CEDEX, France

33 4 76 82 72 80

{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 (MDs) can help to provide nomadic users with relevant and 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 for adaptation purpose by two of the four Multi-Agent Systems (MAS) which constitutes the architecture of PUMAS (the Information and the Adaptation MAS). We also present an example which illustrates the way PUMAS works and takes these facts into account when processing a query.

1. Introduction

Web-based Information Systems (WIS) are systems which allow to collect, structure, store, manage and diffuse information, like traditional Information Systems (IS) do, but over a Web infrastructure. WIS provide their users with complex functionalities which are activated through a Web browser in a hypermedia interface. Nowadays, Mobile Devices (MDs) can be used as devices for accessing distant WIS but also as storage devices for (simple) WIS or applications. Thus, a WIS which executes on MDs allows to access, search and store resources (files) located on these MDs.

However, having to cope with the limited capacities of MDs (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 the delivered information. In order to reach this goal, Multi-Agent Systems (MAS) constitute an interesting approach. The W3C [13] defines an agent as "a concrete piece of software or hardware that sends and receives

messages". These messages can be used for accessing a WIS and for exchanging information. A MAS can be a useful tool for modelling a WIS due to the inherent properties of agents like the knowledge (defined, own and acquired) 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 MDs) in order to satisfy the user's queries. This is the underlying idea of the Mobile Agent concept [8].

Rahwan *et al.* [9] recommend the use of the agent technology in MD applications because agents which execute on the user's MD can inform the systems accessed by the user about her/his contextual information. However, in case of a mobile user, the agent must take into account the fact that the changing location could produce changes in the user's tasks and information needs. Then, the agent also has to be proactive, and has to reason about the user's 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 the Ubiquitous Computing (UC) [13]. Shizuka *et al.* [10] have stressed the fact that Peer to Peer (P2P) computing is one of the potential communicative architectures and technologies for supporting ubiquitous/pervasive computing. Since an agent is an inherent peer - because it can perform its tasks independently from the server and other agents -, we can consider a MAS as a P2P System. P2P systems [10] are characterized by *i*) a direct communication between the peers with no communication needed through a specific server, and *ii*) the autonomy a peer gets for accomplishing some assigned tasks.

Concerning adaptation, a special attention is paid to the user's location in her/his profile. In order to provide the nomadic user only with the more relevant information (i.e. "the right information in the right place at the right time"), Thilliez *et al.* [11] 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 focuses 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 can not 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 adopt. 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 which can not display videos. In that case, the Negotiation vocabulary proposed by Lemlouma [7] can be used for adaptation purposes. It allows describing the user's *MD*, considering the constraints in terms of network and, software and hardware environments.

Many technical and functional aspects have to be considered when designing a *WIS* accessed through *MDs*, especially when addressing the issue of the adaptation of the delivered information to the nomadic user [10] [11]. The goal of our work is to provide nomadic users who access a *WIS* through a *MD* with the more relevant information according to their preferences, but also according to their contextual characteristics and to the features of their *MDs*. In [3], we have defined *PUMAS*, a framework for retrieving information distributed among several *WIS* and/or accessed through different types of *MDs*. The architecture of *PUMAS* is composed of four *MAS* (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. Through *PUMAS*, our final objective is to build and propose a framework which is, beyond the management of accesses to *WIS* through *MDs*, also in charge of performing some adaptation processing over information. Users equipped with *MDs* can use the *PUMAS* central platform in order to communicate together by means of agents executed on their *MDs*, or in order to exchange information (user's contextual information). In our case, users communicate through a *Hybrid P2P system*.

This paper is structured as follows. We first describe in section 2 the architecture of *PUMAS*. We focus on the *pieces of knowledge (facts)* used for adaptation purposes by *PUMAS* agents, especially, those belonging to the *Information* and to the *Adaptation MAS*. In section 3, we present a scenario which shows how *PUMAS* processes a query submitted to the system. An example which illustrates our proposition is given in section 4. We discuss the related work 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 *MAS*, their relations and, the data exchange and the communications they perform in order to achieve the adaptation of the information for the user.

2.1. An overview of the PUMAS architecture

The architecture of *PUMAS* is composed of four *MAS* (see in Figure 1 the logical structure of *PUMAS*):

- The *Connection MAS* provides the mechanisms for facilitating the connection from different types of *MDs* to the system.
- The *Communication MAS* ensures a transparent communication between the *MDs* and the system, and applies a *Display Filter* for displaying the information in an adapted way according to the technical constraints of the user's *MD*. For this, it is helped by agents of the *Adaptation MAS*.
- The *Information MAS* receives the user's query, redirects them to the "right" *IS* (the nearest *IS*, or the one which can answer the user's queries, or 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 the results to the *Communication MAS*.
- The *Adaptation MAS* communicates with the agents of the three other *MAS* in order to provide them with information about the user, the connection and communication features, the *MD* characteristics, etc. The services and tasks of its agents essentially consist in managing specific *XML* files which 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 the analysis of the user's history in the system (last connections, queries, preferences, etc.).

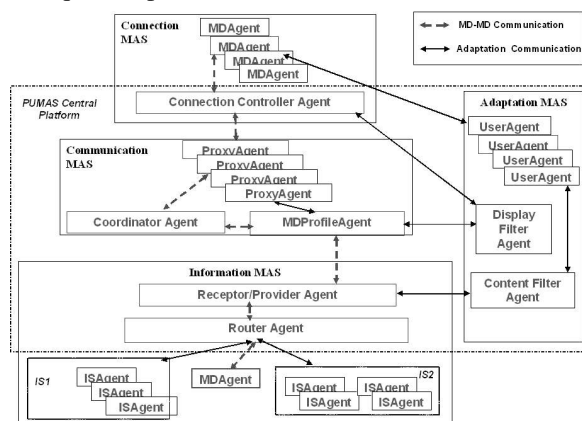


Figure 1. The PUMAS Architecture.

The inherent mobility of the nomadic users is supported by *ubiquitous agents* (e.g., the *MDAs* executed on the user's *MDs* and the *ISAs* executed on the same device than the *WIS*) which retrieve some needed information and which can communicate with

other agents for performing tasks. The *Hybrid P2P Architecture* of *PUMAS* copes with the following issues: security in the applications (security problems inherent to the agent mobility), communication between agents in a point to point or in a broadcast way, management of the agent's status (connected, disconnected, 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 (MDAs)* and one *Connection Controller Agent (CCA)*. The *MDA* is executed on the user's *MD*. Its knowledge is composed of general rules of behavior and characteristics related to the type of *MD* used (e.g., *PDA*) and some specific roles defined according to the application (e.g., this agent is used for transmitting a file). The *MDA* manages a *XML* file (*Device Profile XML* file, located on the user's *MD*) which describes the *MD* characteristics (using *OWL*¹) and, shares this information with the *DisplayFilterAgent* (which belongs to the *Adaptation MAS*) through the *CCA* (the *MDA* sends this file to the *CCA* – executing on the central platform of *PUMAS*– and the latter exchanges this information with the *DisplayFilterAgent*). This file contains some information about requirements of the application, network status, hypermedia files supported by the *MD*, conditions for disconnecting: inactive session for more than *X* minutes, disconnection type (willingly, automatic, etc.), etc.

One *MDA* also manages another *XML* file which describes the characteristics of the user's session (using *OWL*, see Figure 2): who is the connected user (user ID...), when the session begun and what is the connected *MD* (beginning time, *CurrentMD*...). This file will be sent to the *UserAgent* (which belongs 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>
```

Figure 2. User's Session XML file.

The *CCA* gets the user's location and the *MD* type (e.g., *PDA*) from the *User Location XML* file (which contains the physical and logical user's location features) and from the *Device Profile XML* file (which

contains the *MD*'s features). Both files are provided to the *CCA* by the *MDA*.

The *CCA* executes on the central platform of *PUMAS* and gets the user's location and the *MD* type (e.g., *PDA*) from the *User Location XML* file and from the *Device Profile XML* file, respectively. Both files are provided by the *MDA* and locally managed by the *CCA*. The *CCA* serves as an intermediary between the *Connection MAS* and the *Communication MAS*. It also checks the connections established by the users and the agents' status (connected, disconnected, killed, etc.), and links each *MDA* to its corresponding *Proxy Agent (PA)* in the *Communication MAS* (see next section).

The *XML* files (*User Location*, *Session* and *Device Profile XML* files) managed by the *MDA* and the *CCA* have been defined using the extensions introduced by Indulska *et al* [5] to *CC/PP* [13]. These extensions include some user's characteristics like her/his location, application requirements, session features (user, device, application ...) and the *MD*'s profile in order to provide a complete description of the user and her/his *MD*.

2.3. The Communication MAS

This *MAS* has an interface which makes the communication between users transparent and activates the mechanism for displaying the information according to the *MD* features. It is composed by several *Proxy Agents (PAs)*, one *MDProfile Agent (MDPA)* and one *Coordinator Agent (CA)*. These agents execute on the central platform of *PUMAS*.

There is one *PA* for the connection of each *MDA*. Two different users can connect themselves to the system through the same *MD* which leads to two different *PAs* and two different sessions. The main task of a *PA* is to represent a *MDA* within the system. In this case, there are two agents, one *MDA* on the *MD* and one *PA* in the central platform of *PUMAS*.

The *MDPA* has to check the user's profile (according to her/his *MD*) and her/his information needs. In addition, this agent together with the *CA* defines and checks the mechanism for sending, for example, hypermedia data to the user. If the user's request has as results several images, these agents define the order and the number of images to be shown by screen according to the capabilities of the user's *MD*. The *MDPA* also shares information about the specific *MD* features for the user's session with the *DisplayFilterAgent* (of the *Adaptation MAS*).

The *CA* is in permanent communication with the *CCA* in order to verify the connection status of the agent which searches for information. The *CA* knows all the agents connected in the system thanks to *XML* files managed by the *MDA* (through its *PA*). If there are some problems with the *CCA* (e.g. if the *CCA* fails, or if there is a lot of connections...), the *CA* can play the role of the *CCA* up until the problems are fixed. At that moment, the *CCA* and the *CA* must synchronize the

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/>

information about the connected agents and check the current connections.

A more detailed description of the *Connection* and the *Communication MAS* is exposed in [3]. 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 for supporting the adaptation capabilities of *PUMAS*.

2.4 The Information MAS

The *Information MAS* is composed of one *Receptor/Provider Agent (R/PA)*, one *Router Agent (RA)* and one or several *ISAgents (ISAs)*.

The *R/PA* which is located in the central platform of *PUMAS* owns a general view of the whole system. It knows the agents of both the *Communication* and the *Information MAS*. The *R/PA* receives all the requests that are transmitted from the *Communication MAS* and redirects them to the *RA* which is in charge of finding the “right” *IS* in order to execute the query. Once the query has been processed by the *ISAs*, the *R/PA* checks whether the query results take into account the user’s profile (preferences, user’s history...) by means of the *ContentFilterAgent* of the *Adaptation MAS*.

In order to redirect the query to the “right” *IS(s)*, the *RA* applies a strategy which depends on one or several criteria: the user’s location, the peers similarity, the time constraints, her/his preferences, etc. The strategy can lead to the sending of the query to a specific *WIS*, to the sending of the query in a broadcast way and/or to the division of the query in sub-queries, each being sent to one or several *WIS*. The *RA* is also in charge of compiling the 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 of it has to be sent to the *R/PA*.

The *RA* which executes on the central platform of *PUMAS* stores in its *Knowledge Base (KB)* pieces of knowledge (we call *fact* and describe below using *JESS*²) for each *IS*. One fact is made of the characteristics of the *IS* like its name, its managed information, the type of device on which it is executed (e.g., server, *MD*...) and the agent (*ISA*) associated with this *IS* and which can be asked for information. When the *RA* has to redirect the user’s query, it exploits these facts in order to select the *IS*, especially, the *ISAs* to which the sub-queries has to be redirected. The following fact defines an *IS* and is represented by a *JESS* template³:

```
(deftemplate Information_System (slot name) (slot agentID)
(slot device) (multislot information_items))
```

For instance, the following fact defines the *Pharmacy IS* of a hospital. The *IS* is called *PharmacyIS* and it executes on a *server*. *PharmacyISA* is the *ISA* which executes on this *IS*. The *PharmacyIS* contains information about the *medicines* and the *patient’s prescriptions*:

```
(assert (Information_System (name PharmacyIS)
(agentID PharmacyISA) (device server)
(information_items medicines patient’s_prescription)))
```

The location of the *IS* could change, especially if this *IS* run on a *MD*. The *RA* can be informed about the *IS* location changes by means of the *ISAs* which executes on this *IS*.

In order to send the (sub-) queries and analyzing their results, the *RA* must check the user’s preferences (information provided by the *ContentFilterAgent* via the *R/PA*). 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 *ISA* associated with a *WIS* (and which executes on the same device than the *WIS*) receives the user’s (sub-) query from the *RA* and is in charge of searching for information. Once a result for the query is obtained, the *ISA* returns it to the *RA*. An *ISA* can execute the 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 distributed *WIS* located on server(s) but also very simple *WIS* which only rely on some files located on a *MD*. In this last case, one *ISA* may be sufficient to ensure the right functioning of the *Information MAS*. It is worth noting that, in this case, what we call an “*ISA*” is in fact the *MDA* of a *MD* which can play the role of an *ISA* since it has the knowledge required for executing a query on the files stored in the *MD*. In a complex *WIS*, the *ISA* can collaborate with other *ISAs* (if the *WIS* has been developed following the *MAS* paradigm) or with any other *WIS* component to perform the query. In the case of a non *MAS* based *WIS*, our approach only requires that an *ISA* 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 which aims at providing the user with adapted information according to both the user and her/his *MD*. First, the *Content Filter* allows selecting the more relevant information according to the user’s profile defined in the system. Second, the *Display Filter* is applied on the results of the first filter and takes into account the characteristics and technical constraints of the user’s *MD*.

² *JESS* is a rule engine and scripting environment which lets build Java applications that have the capacity to “reason” using knowledge supplied in the form of declarative rules. <http://herzberg.ca.sandia.gov/jess/>

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 “defemplate”. For defining an instance of an unordered fact in *JESS* and storing it into the *JESS KB*, we use the primitive “assert”.

The *Adaptation MAS* is composed of several *UserAgents (UAs)*, one *DisplayFilterAgent (DFA)* and one *ContentFilterAgent (CFA)*. These agents execute on the central platform of *PUMAS*.

Each *UA* manages a *XML file (User Profile XML file, see Figure 3)* which contains personal characteristics of the user (user ID, location, etc.) and her/his preferences (e.g., the user wants only video files). This file is obtained by means of the *MDA* (this file is managed by the *UA* and updated by the *MDA*). There is only one *UA* which represents a user at the same time (even though the user has two sessions at the same time though the same or different *MDs*). Since a user can access the system through several *MDs*, the *UA* communicates with the *MDAs* and the *PAs* (which respectively belong to the *Connection* and the *Communication MAS*) for analyzing and centralizing all the characteristics of the same user. The *UA* communicates with the *CFA* for sending the *User Profile XML file*. When the *CFA* receives this file, it stores this information as facts in its *KB* (this agent manages a register of user's preferences). When the *R/PA* (of the *Information MAS*) asks the *CFA* for the user's preferences, this latter sends it the latest *XML file* received from the *UA*. If the *UA* does not send this file (e.g., there is no user's preferences for the current session), the *CFA* takes into account for this user her/his preferences from previous sessions.

```
<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>
```

Figure 3. User Profile XML file

We can establish that the queries depend on one or several criteria: user's location, her/his history in the system, activities developed during a time period, movement orientation, privacy preferences, etc. A dependency criterion could be defined as:

```
(deftemplate DependencyCriterion (slot userID) (multislot criteria)
(multislot attributes))
```

An example of *Dependency Criterion* which expresses that all of *Doctor Smith's* queries depend on his location, especially when he is in the *North Hospital* could be:

```
(assert (DependencyCriterion (userID Doctor_Smith)
(criteria location) (attributes North_Hospital )))
```

The *DFA* manages a knowledge base which contains general information about features of different types of *MDs* (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 the file formats that are supported by a *Pocket PC hp IPAQ h5550* in different network types is shown as follows. We assume that it can not support videos sent on a *Wi-Fi Network* but it does support several images using *Bluetooth* neither when it is connected through 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 *CFA* manages a knowledge base which contains the preferences, intentions and characteristics of the users. The *User_Preference* fact is composed of the *userID* (which identifies the owner of this preference), the required information (*required_info*) and the complementary information (*complementary_info*). The last one is added to the *User_Preference* definition by the *CFA* which analyzes the queries of the previous sessions (e.g., information frequently asked). This fact is also composed of information describing what and how user would like the answers from the system (to be presented to her/him) and in the case of problems, what and how the system must answer (*list of actions for recovering*). For that, 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 (show, save, transfer file, cancel...) and each action has a list of *attributes*. For instance, the fact which represents the action "show" has for properties the *order*, the *format* and the *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 which 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 can define a *problem* as something which is unexpected, or not wanted to happen when an action is executed, or which is the cause of a failed action execution (e.g., the *MD* can not 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 among a defined list (*incompatibility, unable IS, unable agent*), and the *causes* correspond to a list of causes of this problem (e.g., *MD* can not support a specific format file, network problems, etc.). A fact, which defines the problem

related to a specific user's location which is out of range of a wireless network and prevents to her/him to access to Internet, is:

```
(assert ( problem (name out_range_connection)(type lackofaccess)
(causes userlocatedoutofrange networkoutofservice )))
```

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.

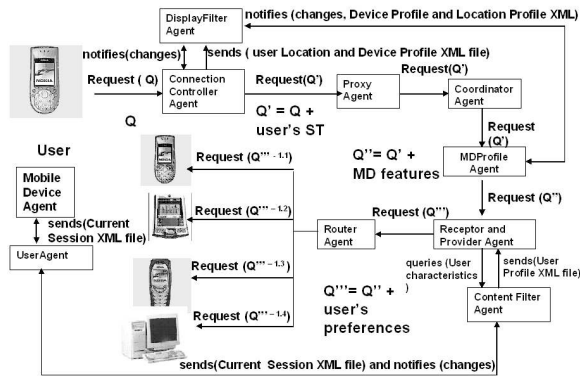


Figure 4. Scenario of sending a query

When a user sends an information query Q (see Figure 4), the MDA sends it to the CCA. Whenever this query is location and time dependent, the CCA introduces the time of connection, the user's location and the user's MD connection characteristics (these latter characteristics are exchanged with the DFA) in query Q which leads to the production of a new query Q' (in Figure 4, $Q' = Q + \text{user's ST}$) which is then sent to the PA. The query passes by the CA and then by the MDPA. The latter adds up into Q' query some features related to the MD; these features are provided by the DFA which have previously learnt them from the previous queries or retrieved them from its knowledge base. The new Q'' query (in the Figure 4, $Q'' = Q' + \text{MD features}$) is sent by the MDPA to the RPA. The RPA adds up at its turn into the Q'' the specific characteristics of the user in the system by requesting the CFA (In Figure 4, $Q''' = Q'' + \text{user's preferences, intentions, history...}$). The RPA sends the Q''' query to the RA which decides (according to the query, the system rules and the fact in its knowledge base) which are the ISAs able to answer. It can send the query to a specific ISA or to several ISAs (e.g., waiting for the first to answer) or, it can divide the query into sub-queries which are sent to one or several ISAs. The scenario in Figure 4, shows for instance that Q''' is divided into $Q'''-1.1$, $Q'''-1.2$, $Q'''-1.3$ and $Q'''-1.4$ which are sent to ISAs executed on a server and different MDs.

When a user $U1$ has an information query for another user $U2$, both equipped with MDs, the query is propagated from the MDA executed on the $U1$'s MD towards the RA which redirects it to the MDA executed on the $U2$'s MD. This $U2$'s MDA changes of role to become an ISA, i.e. the agent in charge of answering the

information query. This change of role is possible because a MDA has the knowledge for managing the information stored in the MD on which it executes and it has the capability of answering the information queries.

4. Example

In this section, we illustrate the process performed by PUMAS agents using the example of a hospital WIS.

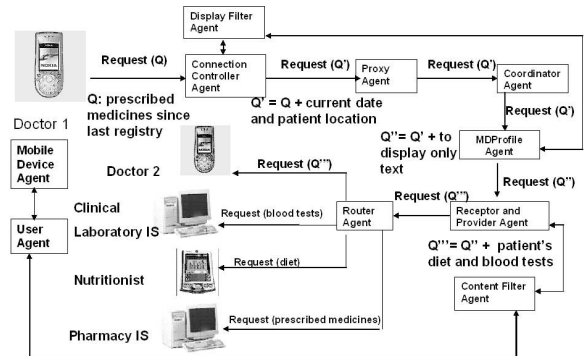


Figure 5. Sending a query in the hospital WIS

Let us suppose that doctors equipped with MDs (e.g. PDA) access to the information system of a hospital which is distributed between several MDs and/or one or several WIS (see Figure 5). Doctors can also receive information according to their location, preferences, technical characteristics of their MDs and considerations about their connection time. For instance, when visiting a patient, doctors with MDs can consult information about her/his clinic history, medical tests, medicines, etc. By indicating the location of the patient (room, floor, bed, etc.) and the current date, the doctor can identify the patient and get her/his personal information. For this, the application on her/his MD must consult the different IS of the hospital – pharmacy, patients, doctors, etc. 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 the room and bed numbers (information about patient's location) while the application gets the date of the system (information about the time). The MDA which executes on the doctor's MD sends the query (who's the patient?). The query is propagated through PUMAS core: it is first transmitted through the CCA, then to the Communication MAS agents (PA, CA and MDPA). The MDPA can add up to the query, the information according to the MD (e.g., this kind of MD can not support graphical format but only text files. then if the doctor asks for the results 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 MDPA can ask the DFA for information about this MD and the MDPA could receive from DFA facts defined as follows:


```
(defacts MDFeature (MDType "Palm Tungsten C")
(feature (type video_not_supported)
(conditions Wi-Fi_Network))
(feature (type several_images)
(conditions Wi-Fi_Network)
(feature (type text)
(conditions Wi-Fi_Network Classical_Network Bluetooth)))) ;
```

Then, the *MDPA* sends the query to the *R/PA* which can add up to the query the 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 *UA* and the *CFA*. The following example corresponds to a doctor's preference: when a doctor says "when asking for a blood test, the system must also provide me with the patient diet and prescribed medicines, I do prefer graphical results but if my MD can not support this format, I shall receive the results in text format", this can be translated in the following fact of the *UA*:

```
(defacts 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 *UA* transfers this information to the *CFA* which stores this fact and sends it to the *R/PA*. The *R/PA* adds this preference to the query and sends it to *RA*. The *RA* receives the complete query and, with the information about the *ISs*, *RA* can split the query in sub-queries and redirects each one towards the appropriated *IS*. The following facts are exploited in this example by the *RA* in order to redirect the (sub-) queries to the *ISAs* of the hospital's *IS*:

```
(assert (Information System (name LaboratoryIS) (agentID
LaboratoryISA) (device server) (information_items test patient's_
test reactive)))
```

```
(assert (Information System (name PatientDietIS) (agentID
DietISA) (machine MD)(patient's_diet nutritionist_appointments)))
```

The *RA* redirects the query to the *ISA* located in the *IS(s)* which manage(s) information about the patients in the hospital. All the queries follow the same path from the *MDA* towards the *RA*. If the doctor wants to know the *last medicines prescribed* to this patient, the *RA* redirects the query to the *ISA* located in the *PharmacyIS*. If the query concerns another *doctor (peer)*, the *RA* redirects the query to the *ISA* 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 *RA* could send the query in a broadcast way or it could decompose the query according to the receiver peer (e.g., queries relates to the heart for the cardiologist...) or according to the defined criteria in the *User Profile XML* file (e.g., if the criterion of query dependency is the *location*, the queries must only be redirected to the doctors at the *same or closed location*

of the sender...). Retrieved information is organized by the *RA* (e.g., the last prescribed medicines, the peer's answers about this patient, etc.) and is returned to the doctor who has sent the query following the inverse path. The different agents have to check the results because, for instance, the doctor may have 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* which supports some graphical format (which constitutes a doctor's preference which can now be satisfied).

Through this example, we can observe the behavior of the *Hybrid P2P Architecture* of *PUMAS*. The core of *PUMAS* centralizes the queries: *i)* it is in charge of the process for obtaining the more relevant information and, *ii)* it is in charge of applying the *Content and Display Filters* for adapting the answers. The main peer characteristics of *PUMAS* agents are illustrated by the fact that first, the agents have the autonomy of connecting to and disconnecting from the system. Second, a *MD* can ask for a communication with a specific *IS* (located on a server or on a *MD*) passing this information as a parameter of the query; the *RA* transmits the query to this specific *IS* which exemplifies an agent to agent communication (e.g., when doctors exchange information about a patient using their *MDs*).

Another advantage offered by *PUMAS* is that it helps a user who does not know which specific *IS* to ask for information to find the more appropriate one(s). The *RA* redirects the query by means of an intelligent analysis of the query and the help of the *ISAs* which achieve an intelligent search inside the different *IS* (pharmacy, laboratory, patients, etc. in our example).

5. Related Works

We present here some agent-based architectures or frameworks for adapting information to the users:

CONSORTS Architecture [6] is based on ubiquitous agents and designed for a massive support of *MDs*. It detects the user's location and defines the user's profile for adapting the information to her/him. The *CONSORTS* architecture proposes a mechanism for defining the relations that hold between agents (communication, hierarchy, role definition...), with the purpose of satisfying user's requests. However, it considers neither the distribution of information between *MDs* (which could improve response time) nor the user's preferences.

The work of Gandon *et al* [4] proposes a Semantic Web architecture for context-awareness and privacy. This architecture supports the 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 to identify the most relevant resources available to answer a query. However, it does not take into account

that the information which can answer a query can be distributed between different sources.

PIA-System [1] is an agent-based personal information system for collecting, filtering and integrating information at a common point, offering access to the information by *WWW*, e-mail, *SMS*, *MMS* and *J2ME* clients. It combines *push* and *pull* techniques in order to allow 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 in 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 the user's needs according to her/his profile, her/his current situation and learning from feedback. However, *PIA-System* only searches information in text format (e.g., documents). It takes into account neither the adaptation of different kinds of media to different *MDs*, nor the user's location.

6. Conclusion

In this paper, we have presented *PUMAS*, a framework based on agents and *P2P* approach. Peers characteristics of *PUMAS* appear in the cooperation developed by the agents in order to store and retrieve the information and in the possibility that two users, equipped with *MDs*, 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 a *Hybrid P2P architecture*. *PUMAS* provides each agent with a mechanism for identifying, authenticating and knowing its peers. This paper has focused on the representation of the *pieces of knowledge* (called *facts*) stored in the 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 the *PUMAS* agents. The search is *intelligent* because is based on the knowledge of the agent and its capability of reasoning. It is also *adaptive* because it takes into account the nomadic user's profile, her/his *MDs*' characteristics and the ubiquitous context features.

Our future work concerns the implementation of each component (*MAS*) of *PUMAS*. We also need to define an extension of the current *ACL* which considers spatio-temporal (contextual) features and a strategy description language, as well as *Query Routing* mechanisms and algorithms [12] for the *RA* in order to propagate the query towards the "*right*" *IS* and to compile the answers.

Acknowledgments. The author *Angela Carrillo Ramos* is partially supported by *Universidad de los Andes*, (Bogotá, Colombia).

7. References

- [1] Albayrak, S., Wollny, S., Varone, N., Lommatzsch, A., and, Milosevic D: Agent Technology for Personalized Information Filtering: The *PIA-System*. In Proc. of *the 20th Annual ACM Symposium on Applied (SAC 2005)* (Santa Fe, New Mexico, USA, March 13-17, 2005). ACM Press, New York, NY (2005), pp. 54-59.
- [2] Carabelea, C., and, Boissier, O: Multi-Agent Platform on Smart Devices: Dream or Reality? In Proc. of *the Smart Objects Conference (sOc'2003)* (Grenoble, France, May 15-17, 2003) (2003), pp. 126-129.
- [3] Carrillo-Ramos, A., Gensel, J., Villanova-Oliver, M., and Martin, H: *PUMAS*: a Framework based on Ubiquitous Agents for Accessing Web Information Systems through Mobile Devices. In proc of *the 20th Annual ACM Symposium on Applied Computing (SAC2005)* (Santa Fe, New Mexico, USA, March 13 -17, 2005) ACM Press, New York, NY (2005), pp. 1003-1008.
- [4] Gandon, F. and, Sadeh, N: Semantic Web Technologies to Reconcile Privacy and Context Awareness. *Journal of Web Semantics*. Volume 1, Issue 3. October 31, 2004. <http://www.websemanticsjournal.org/ps/pub/2004-17>.
- [5] Indulska, J., Robinson, R., Rakotonirainy, A., and K. Henriksen: Experiences in Using *CC/PP* in Context-Aware Systems. In Proc of *Mobile Data Management: 4th Int. Conference (MDM 2003)* (Melbourne, Australia, January 21-24, 2003), LNCS, 2574 (2003), pp. 247-261.
- [6] Kurumatani, K: Mass User Support by Social Coordination among Citizen in a Real Environment. In Proc. of *Multi-Agent for Mass User Support. International Workshop (MAMUS 2003)* (Acapulco, Mexico, August 10, 2003), LNAI, 3012,(2004), pp. 1-16.
- [7] Lemlouma, T. Architecture de Négociation et d'Adaptation de Services Multimédia dans des Environnements Hétérogènes. *Thesis*, Institut National Polytechnique de Grenoble, Grenoble, June 2004.
- [8] Lin, F.C., and, Liu, H.H. *MASPF*: Searching the Shortest Communication Path with the Guarantee of the Message Delivery between Manager and Mobile Agent. In Proc of *Embedded and Ubiquitous Computing (EUC 2004)* (Aizu-Wakamatsu City, Japan, August 25-27, 2004), LNCS, 3207 (2004), pp. 755-764.
- [9] Rahwan, T., Rahwan, T., Rahwan, I., and Ashri, R. Agent-Based Support for Mobile Users Using *AgentSpeak(L)*. In Proc of *Agent-Oriented Information Systems, 5th Int. Bi-Conference Workshop (AOIS 2003)* (Melbourne, Australia, July 14, 2003 - Chicago, USA, October 13, 2003), LNAI, 3030 (2004), pp. 45-60.
- [10] Shizuka, M., Ma, J., Lee, J., Miyoshi, Y., and, Takata, K. A *P2P* Ubiquitous System for Testing Network Programs. In Proc of *Embedded and Ubiquitous Computing (EUC 2004)* (Aizu-Wakamatsu City, Japan, Aug. 25-27, 2004), LNCS, 3207 (2004), pp. 1004-1013.
- [11] Thilliez, M. and, Delot, T: Evaluating Location Dependent Queries Using *ISLANDS*. In Proc of *Symposium on Advanced Distributed Systems (ISSADS 2004)*. (Guadalajara, Mexico, January 25-30, 2004), LNCS, 3061 (2004), pp. 125-136.
- [12] Xu, J., Lim, E., and Ng, W.K: Cluster-Based Database Selection Techniques for Routing Bibliographic Queries. In Proc of *10th Workshop on Database and Expert Systems Applications (DEXA 99)* (Florence, Italy, August 30 – Sept. 3, 1999), LNCS, 1677 (1999), pp. 100-109.
- [13] <http://www.w3.org/TR/webont-req/>

INCA (Investor Network Collaborative Architecture) - A Method in the Madness of Wall Street

Sharad Chandra Sundararajan

IBM

Fishkill, NY, USA

sharads@us.ibm.com

Sandeep Sankarlal

Waters Informatics

Milford, MA, USA

sandeep_s_v@yahoo.com

Ashwani Kumar

MIT

Cambridge, MA

ashwani@mit.edu

Abstract

Folk theories which are based on common, everyday experiences, but not subjected to rigorous experimental techniques, underlie many of our actions [1]. In this paper, we propose an architecture to build an online network of individual investors threaded together in a multi-agent system that exploits aggregate opinion, scientific facts, emotions and common sense to help individual investors speculate with more confidence. The power of the system lies in the interactive mode of operation between the agents and the users, allowing users to post their views blog-style onto a whiteboard and having agents parse these stock logs to re-evaluate stock picks previously made purely based on scientific facts. No system today has such a focused network of investors or a system that gauges the market emotion with continuous user-feedback. A network of this nature has the potential to influence the market considering more investors will have access to the same piece of information.

1. Introduction

The average individual investor makes decisions on trading stocks based on various sources such as the company financials, stock-pundit's analysis, the media, word of mouth etc. And then, there are those who employ brokers and hope for the best while a few others assume the role of a 'contrarian investor' working against popular belief. Irrespective of whether one takes a random or a non-random walk down the investment-lane[2, 3], history has shown that there is no formula that works long enough for it to matter. But, the 'Castle in the air' theory [2] has remained popular for a reason in that, folk-psychology has a tendency to influence the market merely by the strength in its numbers. Yet, no system today exploits such group behavior and no current system takes into account the psychological and emotional factors that in

reality play a significant role in influencing the market. Mob psychology attempts to explain collaborative behavior based on people's psychology and many theorize about this being the root cause of the 1987 crash. So, does the concept of mob-psychology always carry a negative connotation? We don't believe so. There is always strength in numbers, but with incorrect or inaccurate information, the same group can make dangerously incorrect decisions.

In this paper, we propose a strategy and a functional system that will exploit the combination of aggregate opinion, scientific facts, emotions, shared beliefs, common sense investing and analysts' expertise, to help the individual speculate with more confidence. The idea is to grow an online network of individual investors allowing them freedom to post their views on the market to whiteboards, which in this context would be a Stock web LOG (SLOG) and have agents parse these slogs to make sense of them and re-evaluate stock picks previously made purely based on scientific facts.

A network of this nature threaded together by a common vested interest in the market offers some key advantages like: a) Shared knowledge and more awareness; b) Saving time and money; c) Diverse investors offering multiple perspectives on diverse markets (like the international market); d) Developing trust among a group of individuals and boosting morale; e) Most importantly increasing the chances of influencing the market.

For example, the United States used to impose taxes for importing textiles from India and recently the taxes were waived, so the textile market is anticipated to perform well in India. With a network of individuals exchanging stock web logs in a focused domain, such pieces of information can be picked up and used to make more educated decisions on the International market.

This paper discusses a system of versatile agents capable of doing the following:

- Extract the latest company financials and all stock numbers,
- Mine text and filter news articles and forums to gather relevant facts and to sense the overall emotion of the market,
- Read each individual's input comments, do sanity-checking based on some common sense facts and elicit the individual's emotion and tendency towards a stock,
- Group all individual's input along with available expert knowledge and rank the stocks based on time-tested criteria. Provide

advice with corresponding 'reasoning' for the choice

Some of the modules of the architecture like the TextMiner[4] and Common Sense Investing [5] have been studied intensively by other researchers and we hope to use their research as a foundation for some our work.

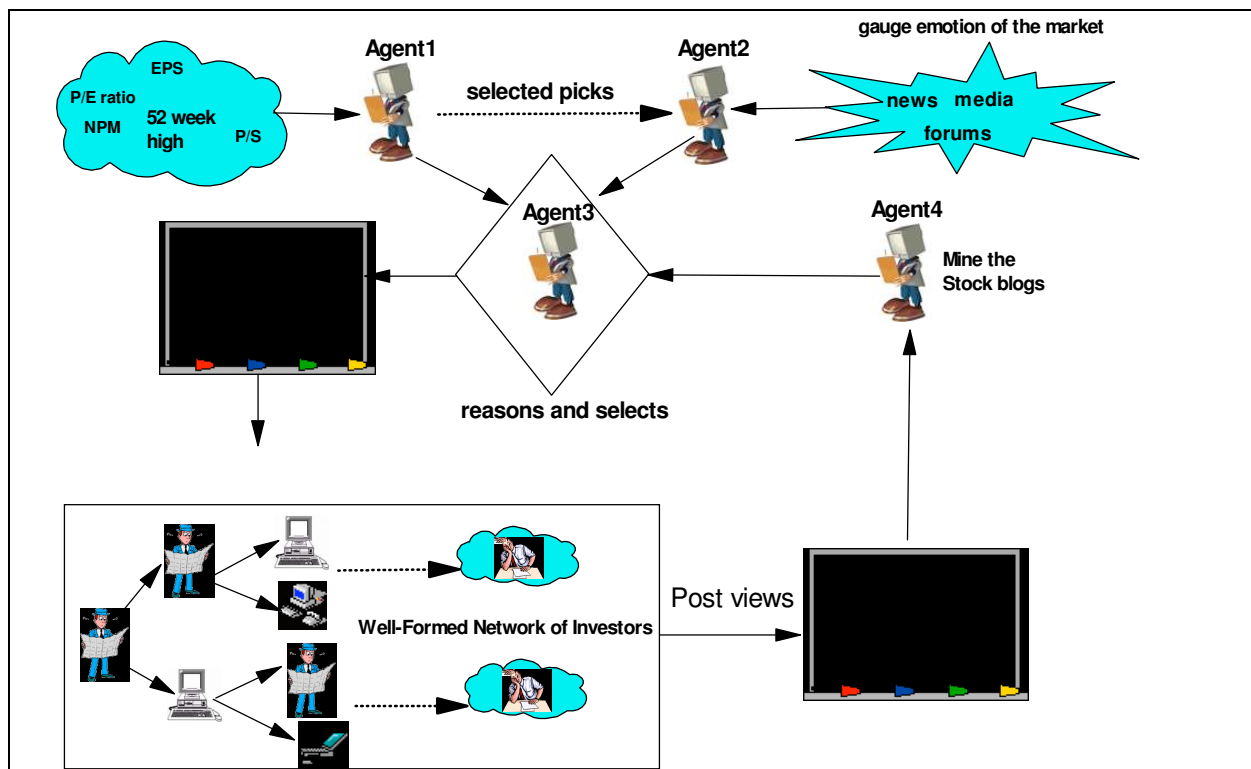


Figure 1: INCA - The overall architecture

StockA						X			
StockB				X					
StockC	X								

Figure 2: The affect of stocks

2. Overall Architecture

The overall design is depicted in Figure 1 and some of the images were taken from [6, 7, 8]. The first and very important part of this system is forming a network [9] of interested individual investors threaded together in a fashion similar to LinkedIn [10] and Friendster [11, 12]. The idea is to have a group of people with a focused and shared interest in making educated decisions on investments. The LinkedIn model is useful to the extent that there is a thread based on trust and experience connecting individuals, but with increasing degrees of separation, the thread can be too thin to be of any use. INCA eliminates any degree of separation by the use of the white board, but uses the concept of a thread to form the network through referrals or recommendations.

There will be four agents collaborating to make the most relevant information available to the users. Agent 1, **InformationExtractor** crawls the web for raw facts like the company financials, price-earnings ratio etc. Agent 2, **AffectSensor** filters forums and news articles [13] to get an idea of what stocks are being talked about and also about the general emotion [14] in the market about specific stocks. Agent 3, **StockAnalyst** which compiles the output data from InformationExtractor and AffectSensor does some reasoning using expert knowledge as well as common sense analysis [5] and ranks allowed to post their views with their own reasoning to a whiteboard and also tag their emotion on the picks as shown in Figure 2, a very visual feedback. The emoticons were taken from [7].

These opinions are posted on a whiteboard which is then parsed and analyzed by Agent 4, the Slogger whose output is fed back to the StockAnalyst. The cycle repeats itself giving the users enough information to make an educated choice. The following section discusses each agent in more detail.

2.1. InformationExtractor

The main objective of this agent is to crawl the web for financial information or company financials in particular in the context of stocks. Several online resources provide valuable data on the different industries, sectors and companies (like nasdaq.com, vanguard.com, fool.com). The history of companies, guru-analysis, pre-market summary and all stock-pertinent information is readily available. InformationExtractor will extract some selected

valuation ratios [15] that dictate the analysts' ratings on the stocks and some of these numbers include:

- P/E (Price to earnings) ratio – Lower P/E ratio represents a better value.
- P/S (Price to Sales) ration – Lower P/S ratio represents a better value
- Price to Free Cash Flow – Lower Price to Free Cash flow represents a better value [16]
- % Owned Institutions – Ideal scenario would be 40% - 75%.
- Earning Per Share – The greater the ratio the better.
- Current ratio – The greater the ratio the better.
- Total Debt to Equity –This ratio should be as low as possible.
- Net Profit Margin – The greater the ratio the better.

In addition, facts that indicate how the stock is trading during the day like the share volume, the day's high and low, and best bid and ask prices etc. are also collected. All this information is used to classify and sort the stocks under different industries and sectors.

2.2. AffectSensor

Although, the output of InformationExtractor may seem sufficient to speculate on the stocks, it is never clear if any one ratio or even a combination of parameters determines how well or bad a company is doing. One of the premises of this paper is that individual investors will benefit from getting an idea of the general feeling in the market about how companies are doing or are predicted to do in the future. This may be folk psychology, but it definitely raises the awareness levels among the users, and in conjunction with the real stock numbers and company statistics, it will allow for a well informed speculation.

This agent is more versatile and powerful than the other agents of the system because of its role to gauge the emotion of the market towards various stocks. AffectSensor filters news articles and forums hunting for opinions and predictions on companies and industries. Warren [13] from CMU, is a multi-agent system for intelligent portfolio management, and offers text classification to elicit the company's financial outlook and their TextMiner [4] does so by classifying articles into the following buckets:

GOOD News articles which explicitly show evidence of the company's healthy financialstatus. e.g.) ... Shares of ABC Company rose 1=2 or 2 percent on the NASDAQ to \$24- 15/16.

GOOD, UNCERTAIN News articles which refer to predictions of future profitability, and forecasts. e.g.) ... *ABC* Company predicts fourth-quarter earnings will be high.

NEUTRAL News articles which mention financial facts but do not provide good or bad aspects. e.g.) ... *ABC* contributes \$ 700 million in stock to its pension plan.

BAD, UNCERTAIN News articles which refer to predictions of future losses, or no profitability. e.g.) ... *ABC* (NASDAQ: *ABC*) warned on Tuesday that Fourth-quarter results could fall short of expectations.

BAD News articles which explicitly show evidence of the company's bad financial status. e.g.) ... Shares of *ABC* (*ABC*: down \$0.54 to \$49.37) fell in early New York trading.

Some common sense reasoning [5, 16] will go into sensing the affect of the text as well. The MIT Media Lab, developed a model to sense textual affect using real world knowledge [14] and this can be exploited to also get the emotion expressed in the articles. Their work also talks about a visual representation of the emotion, the EmpathyBuddy for example [17]. AffectSensor would tag each stock with the market sense using a similar idea.

Input to this agent can either be the output of the InformationExtractor or simply a command to identify important articles or even tabloid news (no smoke without fire).

Another critical component will be to record all the steps in the process of text-mining and reasoning, so that the process of arriving at decisions can be unraveled at any time in the future as a means to troubleshoot flaws or failures. A very good attempt at troubleshooting for the end-user in the domain of E-commerce is the WoodStein[18] project which is an agent that monitors and visualizes user's processes on the web.

2.3. StockAnalyst

There might be young companies with no earnings, which lead to unfavorable ratios and may be tagged as a not-so-good stock to invest in by InformationExtractor and AffectSensor. The valuation ratios may not be applicable to a relatively young company unless viewed in the proper context. On the other hand favorable valuation ratios also do not necessarily guarantee that a company is going to perform as expected. The company's performance might depend on the general state of the economy, the emotional state of the people, and unexpected

occurrences. A classic example is the 1987 stock market crash. There were no indicators of any sort that suggested this crash was coming. In relation to the crash Lope-Markets [19] say "It was the fear among the market investors that a crash was imminent, because the conditions were starting to resemble 1929, the year of the well known crash that ushered in the Great Depression. But the reality was that the economy was still kicking on all cylinders."

StockAnalyst is hence the most critical part of the stock selection. The job of StockAnalyst is to do a detailed analysis of the output of InformationFilter and AffectSensor and use a combination of financial expertise and common sense knowledge [5] to arrive at smart stocks to trade. It is very useful to have tips and negative expertise collected in the past as pointers to avoid pitfalls while making stock selections, especially to assure the users that the economy is not necessarily doing badly because of a few bad eggs or a scenario too similar to an earlier crash. A typical common-sense fact like "high risk => high reward" can be quite useful for the agent to make some calculated risks even to advice stock picks. [20] mentions some very common mistakes (listed below) that investors make. StockAnalyst will either use these pointers internally or prompt to the user to consider:

- Investing in a stock that has been spotlighted in the news recently.
- Buying a stock because it recently had a substantial drop in price.
- Hanging on to a sagging stock waiting for the price to bounce back so you can "get even and get out."
- Falling in love with your stocks.
- Letting your natural disdain for paying taxes overcome your evaluation of the merits of continuing to hold a stock.
- Buying a stock solely because you like the product the firm makes.
- Buying the stock of a great company without considering its price.
- Chasing after initial public offerings.
- Paying too much for growth.
- Buying the stock of a company when you are unable to directly form a judgment on the prospects of the firm.
- Investing in anything you don't fully understand.
- Seeking out high dividend yielding stocks in the belief that the higher the dividend yield, the better.

- Buying a low price-earnings ratio or low price-to-book-value ratio stock without knowing why the ratio is low.

The output of StockAnalyst will be a thoroughly studied set of stocks and will be presented to the users.

2.4. Users

The Users form the heart of the entire system. On seeing the list of stocks on the white board, they follow up with their own views on the agent's stock picks and proceed to create stock web logs.

2.5. Slogger

Blogs are web logs containing periodic time-stamped posts on a common webpage. These have become a craze today, primarily due to the publishing freedom and the power of expression that they offer. Our system attempts to exploit this feature by allowing the participants to create their own blog on stocks and have it be visible to everyone on the network. As per [21] "The major disadvantage is that maintaining a successful blog requires skillful research, professional writing skills and a huge commitment of time and effort. There simply is no such thing as a perfect marketing tool, or an effortless way to build traffic to any site, including blogs." Some of the caveats of blogging are actually addressed by the software agents, as these agents provide the necessary research and skill and also parse the web-logs to make more sense out of them. In addition, the user will be able to mark how they feel about the stocks picked by the agent.

The slogger is an agent that will parse these stock weblogs and elicit the inclination towards particular stocks and this will be passed on to the StockAnalyst which in turn will re-evaluate its previous stock picks. After the stock picks are reevaluated, the slogger posts the picks on the white board. The Slogger also tries to give feedback to the individual users in the network on their posts and how it might have affected the overall selection of stocks. Research has shown that the ability to learn aggregate behavior using network-based recommendation systems is critical to decision-making[22]

3. Sample Scenario: A sample walk through INCA

Note: This analysis is based on real companies and real data but the names of the

companies have been changed. The statistics were taken from [23, 24]

InformationExtractor gets the following information about the stocks A and B from the web. A is an energy company while B deals with public utilities. The two companies have been chosen from different industries just for the sake of diversification. The analysis will be similar even if the industries and sectors are the same.

Table 1: Financial statistics of A and B

Valuation Ratios\ Company name	A	B
P/E	14.7	NA
Price to Sales ratio	2.53	6.38
Price to Free Cash Flow ratio	6.54	NA
% Ownership	67.1	18.9
EPS	1.53	-1.19
Current ratio	0.59	2.92
Total Debt/Equity	0.97	0
Profit Margin	91.60%	-41.08

AffectSensor retrieves the following articles (only snippets shown here) for companies A and B.

News articles about A:

- "Company A reports record results for the fourth quarter and the full year 2004" - [at http://www.okcbusiness.com/news/news_view.asp?newsid=*]
- "Wise Insiders Buy shares of A Again" - [at RealMoney by TheStreet.com]

News articles about B:

- "X's Friends Implicated in B's Trading Scandal" - [at TheStreet.com]
- "Finding Support for B's Stock" - [at RealMoney by TheStreet.com]

AffectSensor using the model from[4] would tag A to be a 'GOOD' stock and B to be 'BAD, UNCERTAIN' one.



		
StockA		X
StockB	X	

Figure 3: Visual sense of Stock Performance

StockAnalyst on receiving the input from InformationExtractor and AffectSensor determines that stock A is a better one, and posts the ranking of the stocks favoring A over B onto the whiteboard. On seeing the new rankings, users post their views. Figure 4 shows a snippet of a real weblog discussing the company A [25]

Is there any fundamental reason to stay with A? Is there more drilling planned in that basin in the near future?

- Joe [5 minutes ago]

A showed great promise back in January and the Big Picture Speculator was all over the breakout. Fundamentally nothing has changed. A Corp. still has a great inventory of Athabasca Basin uranium projects with results pending. There is no recent news that explains the recent high volume sell off. A "promising" Basin project is typically valued in the \$10-\$100 million range. I have a high opinion of A's projects as they didn't have much competition until a year ago.

-Bill [25 minutes ago]

Any comment on the mining/metal meltdown today? Think it is profit taking, opportunities perhaps to buy more.

-Karl [2 hours ago]

I haven't heard about this company. I am not sure of the prospects.

-Jim [5 hours ago]

Here are the highlights from A Resources' recent earnings report. Earnings per share of 92 cents and cash earnings per share of \$1.53; Rough oil sales revenue increased by \$157-million over the prior year;Credit facility refinanced from project loan to a combined secured term loan and revolving credit facility; Acquired a 51-per-cent controlling interest in XYZ Inc.; and Dividend policy implemented and share repurchase plan approved.

-Tracy [1 day ago]

Figure 4: A real stock Log by users (discussing company A)

Slogger goes through the above web logs and finds out the general consensus and tags the stock (A)

as a 'STRONG BUY' and passes this information to StockAnalyzer. StockAnalyzer in turn posts this message back on the white board.

We believe the design will evolve with time to address any noise in the system such as lying and mistrust due to conflicting interests within the network [26]. We do not mean to trivialize the impact of manipulation in the network, but would rather not shy away from the system due to such issues. It is possible that users may join the network with an ulterior motive of trying to mislead the group based on their own bias. But this is where technology intervenes to reduce if not eliminate such noise. In our architecture, the agent 'StockAnalyst' makes decisions on top quality stock picks based on real company financials. Now, if the Stock Logs indicate otherwise by stating that some relatively low-performing stocks are actually the best to invest in, StockAnalyst will point out the discrepancy providing an explanation based on the blogger's comments. The individual investor always has the final say, the main advantage now being that he makes an educated pick.

4. Conclusion

We are currently in the process of building this system in order to evaluate its effectiveness in the real world and plan to take more case studies through INCA to validate our hypothesis. There is a lot of potential in building such a network that is sensitive to the market emotion and users' feedback in addition to using both expert financial knowledge as well as common sense reasoning. We hope to conduct more research in this area. The current design is targeted to the average novice investor and we hope to extend it to be useful for a wider expert audience.

5. References

- [1] Description of Folk-Psychology,Wikipedia. Retrieved March 5, 2005 from http://en.wikipedia.org/wiki/Folk_psychology
- [2] Malkiel, B. G. A Random Walk Down Wall Street, W.W. Norton & Company Inc., New York, 2003.
- [3] Ritter R.Jay, Behavioral Finance Pacific-Basin Finance Journal Vol. 11, No. 4, (September 2003) pp. 429-437.
- [4] TextMiner:Text Classification for Intelligent Agent Portfolio Management, Retrieved February 20 from http://www-2.cs.cmu.edu/~softagents/text_miner.html

- [5] Kumar, A., Sundararajan, S. and Lieberman, H. Common Sense Investing: Bridging the Gap Between Expert and Novice. Proceedings of Conference on Human Factors in Computing Systems (CHI '04). Vienna, Austria.
- [6] Computer icons, Retrieved March 6, 2005 from <http://www.iconbazaar.com/computer/pg05.html>
- [7] Icons Retrieved March 4, 2005 from <http://www.animationfactory.com/animations>
- [8] Clipart Retrieved March 4, 2005 from <http://classroomclipart.com/cgi-bin/kids/imageFolio.cgi?direct=Clipart/Computer>
- [9] Buskens, Vincent and Jeroen Weesie (2000) Cooperation via social networks. Reprinted from *Analyse & Kritik* 22 (2000): 44-74.
- [10] Building Professional networks Retrieved March 2, 2005 from <https://www.linkedin.com/>
- [11] Building Social networks, Retrieved March 2, 2005 from <http://www.friendster.com>
- [12] Boyd, danah. "Friendster and Publicly Articulated Social Networks." Conference on Human Factors and Computing Systems (CHI 2004). Vienna: ACM, April 24-29, 2004.
- [13] Seo, Y. W. Giampapa, J. and Sycara, K. Financial news analysis for intelligent portfolio management. Technical Report CMU-RI-TR-04-04, Robotics Institute, Carnegie Mellon University, 2004.
- [14] Liu, H., Lieberman, H. and Selker, E. A Model of Textual Affect Sensing using Real-World Knowledge. Proceedings of the ACM International conference on Intelligent User Interfaces (IUI '03) (January 12-15, 2003, Miami, FL, USA). ACM 2003, ISBN 1-58113-586-6, pp. 125-132.
- [15] Coval, Chris, Fundamental Analysis of the Stock Market, Retrieved March, 8 2005 from http://www.incometrader.com/Fundamental_Analysis.htm
- [16] Bogle, J. C. Common Sense on Mutual Funds, John Wiley & Sons Inc., New York, 1999.
- [17] Liu, H., Lieberman, H., and Selker, T. Automatic Affective Feedback in an Email Browser. MIT Media Lab Software Agents Group Technical Report. November, 2002.
- [18] Lieberman, H. and Wagner, E. End-User Debugging for Electronic Commerce. ACM Conference on Intelligent User Interfaces, Miami Beach, January 2003.
- [19] The 1987 Stock Market Crash, Retrieved March 1, 2005 from <http://www.lope.ca/markets/1987.html>
- [20] Common Mistakes Investors Make, Retrieved March 10, 2005 from <http://www.aaii.com/>
- [21] Ochman, BL, Straight Talk About Blogs, Retrieved April 15, 2005 from <http://www.frugalmarketing.com/dtb/blogs.shtml>
- [22] Dan Ariely, John G. Lynch and Manny Aparicio (2004), "Learning by Collaborative and Individual-Based Recommendation Agents," *Journal of Consumer Psychology*, 14 (1&2), 81- 94.
- [23] Advisor FYI Definitions and Company Financials, Retrieved March 10, 2005 from <http://moneycentral.msn.com/>
- [24] Companies performance, Retrieved March 10, 2005 from <http://www.forbes.com>
- [25] Sample blog, Retrieved March 11, 2005 from <http://www.globeofblogs.com>
- [26] Evans, D., Heuvelink, A., and Nettle, D., The evolution of optimism: a multi-agent based model of adaptive bias in human judgement. In Proceedings of the AISB'03 Symposium on Scientific Methods for the Analysis of Agent-Environment Interaction, University of Wales, pp 20-25, 2003

An Agent-Based Meta-Level Architecture for Strategic Reasoning in Naval Planning

Mark Hoogendoorn¹, Catholijn M. Jonker³,
Peter-Paul van Maanen^{1,2}, and Jan Treur¹

¹Vrije Universiteit Amsterdam,
Dept. of Artificial Intelligence
De Boelelaan 1081a,
1081 HV Amsterdam,
The Netherlands
{mhoogen, pp, treur}@cs.vu.nl

²TNO Human Factors,
Dept. of Information
Processing,
P.O. Box 23,
3769 ZG Soesterberg,
The Netherlands

³Radboud University Nijmegen,
Nijmegen Institute for
Cognition and Information,
Montessorilaan 3,
6525 HR Nijmegen,
The Netherlands
C.Jonker@nici.ru.nl

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 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.

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, which results in an unmeant chain of events if left alone. Strategic reasoning in a planning context can occur both in *plan generation* strategies (cf. [15]) 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. [10]) is presented for planning, extended with a strategic reasoning level. Besides the introduction of an 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.

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 behavior 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 [2]. For a comparison of DESIRE with other agent-based modeling techniques, such as GAIA, ADEPT, and MetateM, see [13, 11]. The top-level of the system is shown in Figure 1 and consists of the *ExternalWorld* and the *Agent*. The *ExternalWorld* generates observations which are forwarded to the *Agent*, and executes the actions that have been determined by the

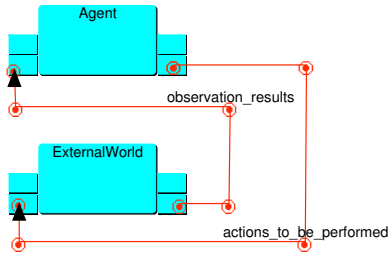


Fig. 1. Top-level architecture

Agent. The composition of the Agent is based on the generic agent model described in [3] of which two components are used: WorldInteractionManagement and OwnProcessControl, as shown in Figure 2. WorldInteractionManagement takes care of monitoring the observations that are received from the ExternalWorld. In case these observations are consistent with the current plan, the actions which are specified in the plan are executed by means of forwarding them to the top-level. Otherwise, evaluation information is generated and

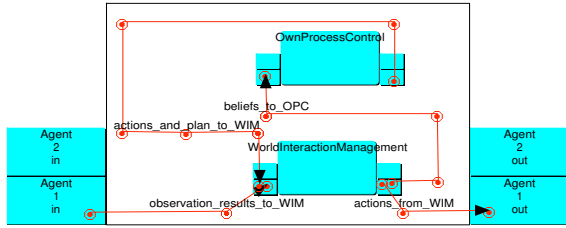


Fig. 2. Agent architecture

forwarded to the OwnProcessControl component. Once OwnProcessControl receives such an evaluation it determines whether the current plan needs to be changed, and in case 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 Figure 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 case no plan can be selected in PlanSelection this information is forwarded to the StrategyDetermination component. StrategyDetermination reasons on a meta-level (the input is located on a higher level as well as the output as shown in Figure 3), getting input by translating beliefs into reflected beliefs and by means of receiving the status

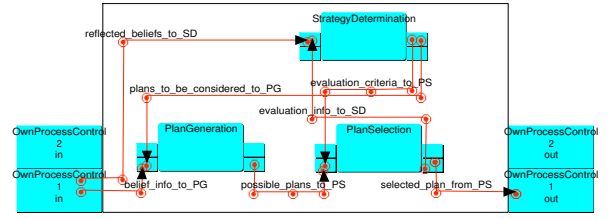


Fig. 3. Components within OwnProcessControl

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 time frame T is assumed which is linearly ordered. A trace or trajectory γ over a state ontology Ont and time frame T is a mapping $\gamma : T \rightarrow STATES(Ont)$, i.e., a sequence of states $\gamma(t \in T)$ in $STATES(Ont)$. The set of all traces over state ontology Ont is denoted by $TRACES(Ont)$. Depending on the application, the time frame 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 $DYNPROP(\Sigma)$ 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 $state(\gamma, t, input(c))$. Analogously $state(\gamma, t, output(c))$ and $state(\gamma, t, internal(c))$ denote the output state, internal state and external world state.

These states can be related to state properties via the formally defined satisfaction relation \models , comparable to the Holds-predicate in the Situation Calculus: $state(\gamma, t, output(c)) \models p$ 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 $state(\gamma, t, output(c)) \models p$ are shortened to $output(c) \models p$.

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 non-negative real numbers. In the *leads to* language $\alpha \rightarrow_{e, f, g, h} \beta$, 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 part of OwnProcessControl within the agent as 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:SITUATION, pos)
and conditionally_allowed(S:SITUATION, P:PLAN)
then candidate_plan(P:PLAN)
```

Stating that in case 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) fleet morale criterion. In this scenario it is assumed that a weighed 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 weighed sum can be taken over time. To obtain the observations, for each candidate plan the consequence 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 case 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 labeled 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 case 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; (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 weighed 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 criterion. Morale is important in the considerations as troops with a good morale are much more likely to win compared to those who do not have a good morale. 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.

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):

$$\text{evaluation_value}(P:\text{PLAN}) = \alpha * \text{mission_success_value}(P:\text{PLAN}) + \beta * \text{safety_value}(P:\text{PLAN}) + \gamma * \text{fleet_morale_value}(P:\text{PLAN})$$

where all values and degrees are in the interval [0,1], and $\alpha + \beta + \gamma = 1$. 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 β 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 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 Figure 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:

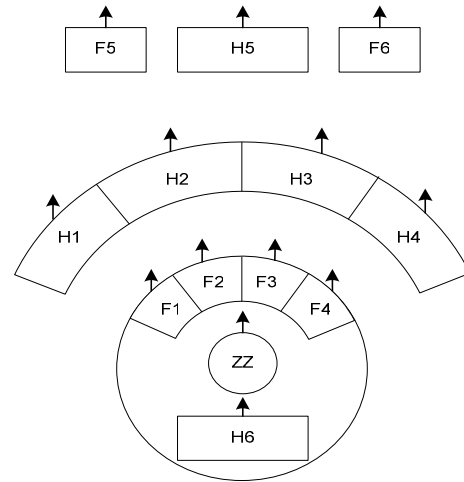


Fig. 4. Scenario for meta-reasoning

1. **Continue sailing.** Leave the ship behind. The safety of the main fleet will therefore be maximal, however 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, however 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.

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 Figure 5. The trace, a temporal description of chains of events, describes the decision making process of the agent which pays 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

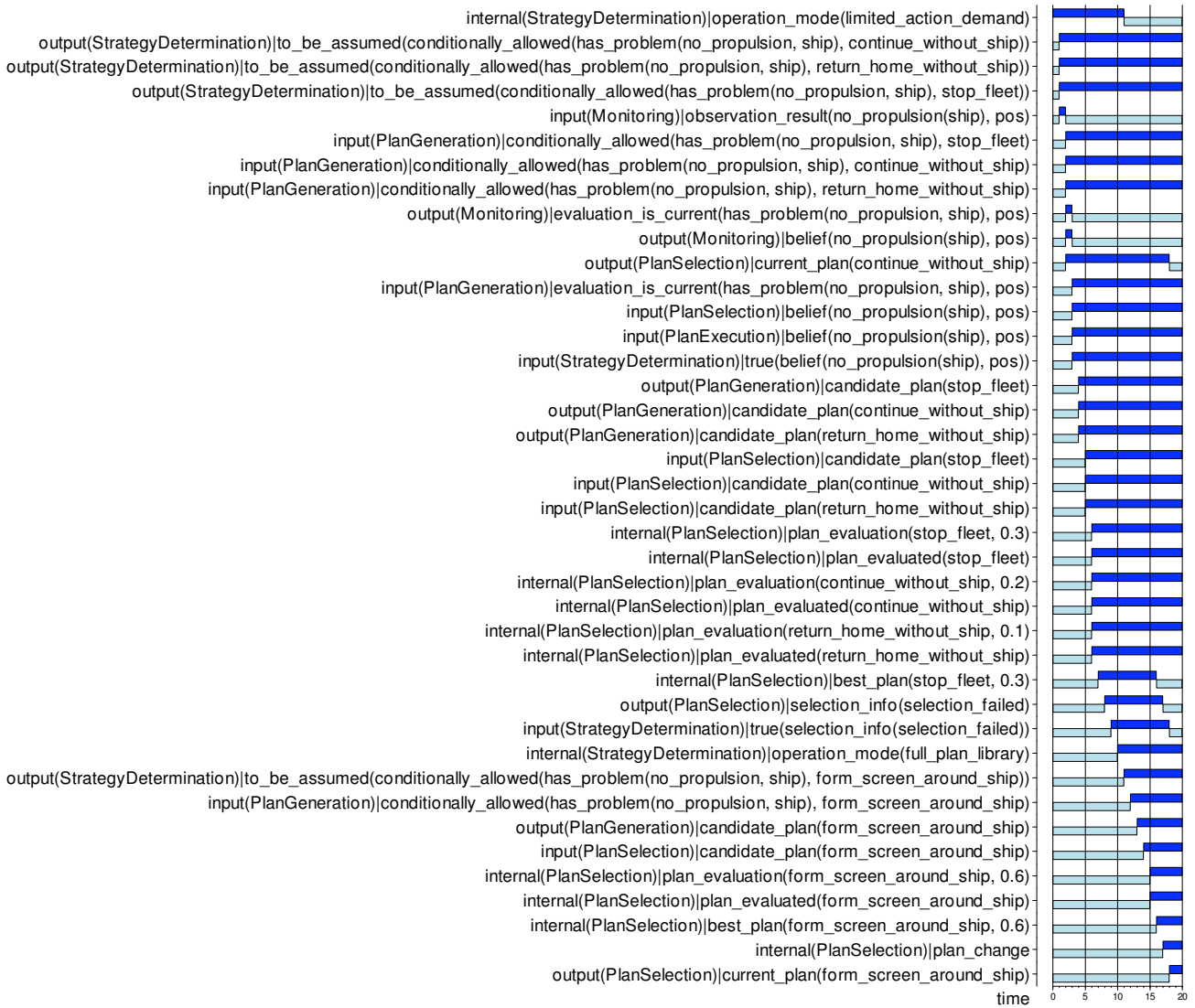


Fig. 5. Trace of the total steam failure simulation

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 case of a black box the atom is false during that period, in the 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 which 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 info 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. Below the verification of these properties in the trace are shown. The properties are independent from the specific scenario and should hold for every scenario for which the agent-based meta-level architecture presented in Section 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. In semi-formal notation:

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:

$$\forall t \ [[\forall O:OBS, S:SIGN$$

$$[state(\gamma, t, output(Monitoring)) \models belief(O, S)$$

$$\Rightarrow \exists t2 \geq t \ state(\gamma, t2, input(StrategyDetermination)) \models true(belief(O,S))]]$$

$$\& [\forall SI:SEL_INFO [state(\gamma, t, output(PlanSelection)) \models selection_info(SI)$$

$$\Rightarrow \exists t2 \geq t \ state(\gamma, t2, input(StrategyDetermination)) \models$$

$$true(selection_info(SI))]]]$$

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:

$$\forall t, S:SITUATION, P:PLAN [state(\gamma, t, output(StrategyDetermination)) \models$$

$$to_be_assumed(conditionally_allowed(S, P))$$

$$\Rightarrow \exists t2 \geq t \ state(\gamma, t2, input(PlanGeneration)) \models conditionally_allowed(S, P)]$$

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:

$$\forall t, t2 > t, S:SITUATION, P1:PLAN, P2:PLAN$$

$$[[state(\gamma, t, output(Monitoring)) \models evaluation(exception(S), pos) \&$$

$$state(\gamma, t, output(PlanSelection)) \models current_plan(P1) \&$$

$$state(\gamma, t2, output(PlanSelection)) \models current_plan(P2) \& P1 \neq P2$$

$$\& \ state(\gamma, t2, internal(StrategyDetermination)) \models$$

$$to_be_assumed(preferred_plan(S, P2))$$

$$\Rightarrow \exists t' [t' \geq t \& t' \leq t2 \& state(\gamma, t', output(PlanSelection)) \models$$

$$selection_info(selection_failed)]]]$$

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. Formal:

$$\forall t, t2 \geq t, P:PLAN [[state(\gamma, t, output(PlanSelection)) \models current_plan(P) \&$$

$$state(\gamma, t2, output(PlanSelection)) \models current_plan(P)]$$

$$\Rightarrow \exists t', S:SITUATION [t' \geq t \& t' \leq t2 \& state(\gamma, t',$$

$$output(Monitoring)) \models evaluation(exception(S), pos)]]]$$

This property is again satisfied for the given trace.

7. Discussion

This paper presents an agent-based architecture for strategic planning (cf. [15]) for naval domains. The architecture was designed as a meta-level architecture (cf. [10]) with three levels. The interaction between the levels in this paper is modeled by reflection principles (e.g., [1]). The dynamics of the architecture is based on a multi-level

trace approach as an extension of what is described in [6]. 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.

A meta-level architecture for strategic reasoning in another area, namely that of design processes is described in [4]. This architecture has been used as a source of inspiration for the current architecture for strategic planning. In other architectures, such as in PRS [5], meta-level knowledge is also part of the system, however 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. Finally, this paper addressed resource-bounded situations. In [12] an overview is presented of models for human behavior that can be used for simulations. Similar to research done in other agent-based systems using the DESIRE framework [2], 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.

Acknowledgements

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.

References

1. Bowen, K. and Kowalski, R., Amalgamating language and meta-language in logic programming. In: K. Clark, S. Tarnlund (eds.), *Logic programming*. Academic Press, 1982.
2. Brazier, F.M.T., Dunin Keplicz, B., Jennings, N., and Treur, J., DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. *International Journal of Cooperative Information Systems*, vol. 6, 1997, pp. 67-94.
3. Brazier, F.M.T., Jonker, C.M., and Treur, J., Compositional Design and Reuse of a Generic Agent Model. *Applied Artificial Intelligence Journal*, vol. 14, 2000, pp. 491-538.
4. Brazier, F.M.T., Langen, P.H.G. van, and Treur, J., Strategic Knowledge in Design: a Compositional Approach. *Knowledge-based Systems*, vol. 11, 1998 (Special Issue on Strategic Knowledge and Concept Formation, K. Hori, ed.), pp. 405-416.
5. Georgeff, M. P., and Ingrand, F. F., Decision-making in an embedded reasoning system. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pages 972-978, Detroit, MI, 1989.
6. Hoek, W. van der, Meyer, J.-J.Ch., and Treur, J., Formal Semantics of Meta-Level Architectures: Temporal Epistemic Reflection. *International Journal of Intelligent Systems*, vol. 18, 2003, pp. 1293-1318.
7. Jonker, C.M., and Treur, J., A Compositional Process Control Model and its Application to Biochemical Processes. *Applied Artificial Intelligence Journal*, vol. 16, 2002, pp. 51-71.
8. Jonker, C.M., and Treur, J. Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. *International Journal of Cooperative Information Systems*, vol. 11, 2002, pp. 51-92.
9. Jonker, C.M., Treur, J., and Wijngaards, W.C.A., A Temporal Modelling Environment for Internally Grounded Beliefs, Desires and Intentions. *Cognitive Systems Research Journal*, vol. 4, 2003, pp. 191-210.
10. Maes, P, Nardi, D. (eds), *Meta-level architectures and reflection*, Elsevier Science Publishers, 1988.
11. Mulder, M, Treur, J., and 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, pp. 193-207.
12. Pew, R.W. and Mavor, A.S.. *Modeling Human and Organizational Behavior*, National Academy Press, Washington, D.C. 1999.
13. Shehory, O., and Sturm, A., Evaluation of modeling techniques for agent-based systems, In: *Proceedings of the fifth international conference on Autonomous agents*, Montreal, Canada, May 2001, pp. 624-631.
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, pp. 179-186.
15. Wilkins, D.E., Domain-independent planning Representation and plan generation. *Artificial Intelligence* 22 (1984), pp. 269-301.

Design Options for Subscription Managers

Aloys Mbala
RMIT University
Melbourne, Australia
alloys@cs.rmit.edu.au

Lin Padgham
RMIT University
Melbourne, Australia
linpa@cs.rmit.edu.au

Michael Winikoff
RMIT University
Melbourne, Australia
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 WSDL¹ and OWL-S [10], as well as work on distributed search algorithms and architectures such as peer-to-peer systems [11]. A common approach, even in peer-to-peer systems, is to have some specialised agents (or services) which assist providers and requesters to find one another. These are variously called *yellow page agents* [1], directory facilitators², brokers [4], and match-makers [12] with the term *middle-agent* being used to characterise these kinds of agents. UDDI (Universal Description, Discovery and Integration) directories³ are one standard instantiation of such a facility while FIPA (Foundation for Intelligent Physical Agents) Directory Facilitators are another.

1 <http://www.w3.org/TR/wsdl>

2 <http://www.fipa.org/specs/fipa00023/SC00023K.html>

3 <http://www.uddi.org>

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 [9], 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 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 protocols⁴.

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 [5].

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

4 e.g. <http://www.fipa.org/specs/fipa00035/>

provision of information on available relevant providers to requesters is a key difference between our work and event notification systems such as Siena [2] or NaradaBrokering [7], which do not provide requesters with information on changes to available providers⁵.

In this paper we explore design options for “subscription manager” middle agents which 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 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 [6], and others such as LARKS [13] and Infosleuth [3].

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.

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.

2. The Interaction Models

Service Discovery frameworks can be categorised 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 which 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 [8] 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. [7, 2, 11, 8]). 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, 5, 12, 14] has compared different styles of middle-agents and concluded that *Matchmakers* which 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 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.

⁶ A single agent can be both a provider and a requester, but for the purpose of this work we consider them separately.

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 [12] mentions a monitoring capability, although very little detail is given⁷. The notion of facilitator defined by Finin et. al. [6] 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 trade-offs.

Figure 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).

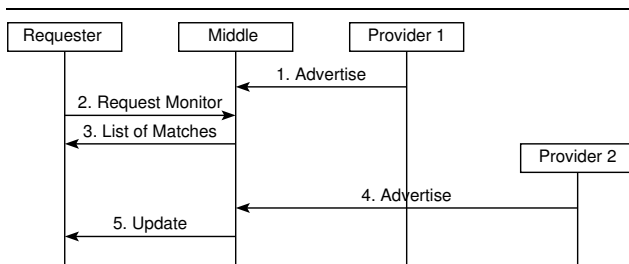


Figure 1. The discovery mechanism

However, this figure is incomplete as it focuses 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 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.

⁷ The notion of “monitor” vs. “single shot” match-making is mentioned on page 42 of [12].

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 figure 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 one⁸.

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

⁸ An additional form would be subscribe to N .

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.

3. Analysis

In this section we analyse the alternative design choices for a Subscription Manager middle-agent. The analysis focuses 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, and therefore we also use the size of the messages to estimate the amount of network traffic.

The analysis in this section uses the following terms:

- 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.

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.

3.1. Adding an Agent

3.1.1. 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 with the middle-agent its interests, (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.

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 with the middle-agent its interests, (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 is $k + P_S$ (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 which providers to subscribe to). 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 summarised in figure 2. In all cases informing the other agent takes a single additional message of size equal to the number of desired providers.

	Middle Subscribes	Requester Subscribes
All providers	$1 + P_F$ $(k + P_F)$	$2 + P_F$ $(k + 2P_F)$
P_S providers	$1 + P_S$ $(k + P_S)$	$2 + P_S$ $(k + kP_F + P_S)$

Figure 2. Adding a requester (message size analysis is in brackets)

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).

3.1.2. 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 with the middle-agent its capabilities, (2) the middle agent sends a message back to the provider

	Middle Subscribes	Requester Subscribes
All providers	2 $(k + R_F)$	$1 + 2R_F$ $(k + 2R_F)$
typical P_S providers	1 (k)	1 (k)
max. P_S providers	2 $(k + R_S)$	$1 + R_F + R_S$ $(k + R_F + R_S)$

Figure 3. Adding a provider (message size analysis is in brackets)

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 summarised in the top row of figure 3. Informing the requester (if the Subscription Manager subscribes) takes an additional R_F messages of size 1.

The bottom two rows of figure 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 figure 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 ping⁹. 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 follows¹⁰. Firstly, there are P messages to the providers. Secondly, there are P_D responses, one for each departed agent¹¹, 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 ($P_D \times R_F$ messages¹²), 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 realise 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

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 middle-ware (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 $P_D \times R_S$

Who pings?	Number of messages	+ Implicit pings
Middle agent	$P + P_D$ ($P + P_D + P_D R_S$)	N/A
Requester agents	$P R_S + P_D R_S$ ($P R_S + 2 P_D R_S$)	$2 P_D R_S$ ($3 P_D R_S$)
Improved	$P R_S + 2 P_D$	$2 P_D + P_D R_S$

Figure 4. Monitoring provider liveness (bracketed formulae include informing)

(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. $(P - P_D) \times R_S + P_D \times (2 + R_S) = P \times R_S + 2 P_D$.

Note that this more efficient, but more complex, approach requires that the middle-agent have a record of subscriptions (otherwise it is more expensive: replace 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.

One potential further saving in having liveness monitoring be done by requesters is that it becomes possible to use “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 optimisation 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 optimisation 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 $P_D \times (2 + R_S) = 2 P_D + P_D R_S$ messages.

This analysis is summarised in figure 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 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. 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 3.1.1.

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 all¹³. This requires that the interface allows four¹⁴ kind of requests: *single-match* (requester subscribes), *ongoing-match* (requester subscribes), *subscribe-one* (Subscription Manager subscribes 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 it

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.

relieves the Subscription Manager of a centralised 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 actually requires fewer messages ($2P_D + P_D R_S$ compared with $P + P_D + P_D R_S$). However, this requires a more complex mechanism, shifts the responsibility for a crucial infrastructure task onto the requesters (which is not practical in an open system), and assumes that implicit pings completely eliminate pinging of live agents and that requester agents need to be informed of departed providers¹⁵. Further, even in the best case, the savings by having requester agents monitor provider liveness are not significant. Therefore, we recommend that monitoring of provider liveness be done by the Subscription Manager.

Due to space limitations we are unable to provide a full interface specification of the Subscription manager, but it should be evident from the above discussion.

5. Conclusion

We 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.

Areas for future work include investigating ways of structuring a network of middle-agents, carrying out experimental work, and looking at how often agents should be ‘pinged’ given a particular rate of agent departure.

Acknowledgements

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.

¹⁵ If requesters are not required to be informed of departed providers, then having middle-agents monitor providers requires fewer messages ($P + P_D$) and in this case having requesters monitor is more efficient if $P_D(1 + R_S) < P$.

References

- [1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin, Germany, 2004.
- [2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical Report CU-CS-863-98, University of Colorado, Department of Computer Science, 1998.
- [3] A. Cassandra, D. Chandrasekara, and M. Nodine. Capability-based agent matchmaking. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 201–202. ACM Press, 2000.
- [4] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Fifteenth International Joint Conference on Artificial Intelligence*, pages 578–583. Morgan Kaufmann, August 1997.
- [5] K. Decker, M. Williamson, and K. Sycara. Matchmaking and brokering. In *2nd International Conference on Multi-Agent Systems (ICMAS 1996)*. MIT Press, 1996.
- [6] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM Press, 1994.
- [7] G. Fox and S. Pallickara. 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)*, pages 353–359, 2002.
- [8] N. Gibbins and W. Hall. Scalability issues for query routing service discovery. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pages 209–217, 2001.
- [9] I. Mathieson, S. Dance, L. Padgham, M. Gorman, and M. Winikoff. An open meteorological alerting system: Issues and solutions. In V. Estivill-Castro, editor, *Proceedings of the 27th Australasian Computer Science Conference*, pages 351–358, Dunedin, New Zealand, 2004.
- [10] M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara. A broker for OWL-S web services. In *First International Semantic Web Services Symposium*, 2004.
- [11] C. Schmidt and M. Parashar. A peer-to-peer approach to web service discovery. *World Wide Web Journal*, 7(2):211–229, 2004.
- [12] K. Sycara. Multi-agent infrastructure, agent discovery, middle agents for web services and interoperation. In *Multi-Agent Systems and Applications, LNAI 2086*, pages 17–49. Springer-Verlag, 2001.
- [13] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5(2):173–203, 2002.
- [14] H. C. Wong and K. Sycara. A taxonomy of middle-agents for the internet. In *4th International Conference on Multi-Agent Systems (ICMAS 2000)*, pages 465–466. IEEE Press, 2000.

Supporting Program Indexing and Querying in Source Code Digital Libraries

Yuhanis Yusof and Omer F. Rana
Cardiff 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 into a library and facilitate search over this digital library. To facilitate users, we propose the usage of agents in indexing and querying program source code. This paper describes agent roles in building index file for Java programs and users query 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 enable 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 rationale for decisions throughout the life of a project. Until recently, data from these repositories was 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 this 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 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 make 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 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 multi-agent system in supporting program indexing and querying in source code digital libraries.

2. Overview

Software reuse is an approach to developing systems where artifacts that already exists are used again. Software artifacts vary from software components to analysis models. A major problem in software development today occurs when different artifacts 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 or formal documentations are often not modified to reflect these changes. Therefore, being the most updated information source of a software, program source code is used by software developers in program understanding. In this paper, we concentrate exclusively on reusing 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 where it describes how 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.

2.1. 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 mainly focused on categorizing the programming language used or source code achieve [14]. Ugurel et al. [19] 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 [15] have produced a framework which 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, focus on the retrieval of software component. Ostertag et al. [10] classified components retrieval approaches by 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 therefore are 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 component from repositories, but the approach is labor intensive. The faceted classification scheme for software reuse proposed by Prieto-Daz [12] relies on facets which 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 labor intensive, and often intended for use in a specific application domain. Sugumaran [17] 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 reuse, researchers have also been investigating component retrieval based on formal specifications [11, 16, 9, 8]. Mili et al. [8] 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 specification, 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. 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 [8] and Schumann [16] 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 are The University of Michigan Digital Library (UMDL) [3], The Multimedia Electronic Documents (MeDoc) system [2] and the ZUNO Digital Library (ZUNODL) [4] – 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`, `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 software components from the digital library. 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 In-

ternet [18, 7, 6, 3].

2.2. 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, as search engines such as `Google.com` or `Altavista.com` only provide support for formulating a query based on keywords and phrase. The search process utilised in `SourceForge` 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 try to retrieve suitable programs based on its program structure and design pattern. Similar to indexing journal articles (author, title and year representing important features of an article), Java program structure is used to represent each of our stored programs. We include *classes*, *comments*, *identifiers*, *packages* and *import statements* as components of the program structure. Each of these components plays significant role in defining 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 good practice to name programs or function modules according to its functionalities [13], identifiers are used to represent functionality of the program. Using Java program structure as the basis to define our design pattern rules, we try to identify general design implemented in the programs which 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 making full use of the code (cut and paste), developers using our digital libraries are provided with relevant knowledge relating to the problem.

In this research, each program and user query submitted to the digital library is represented by an index file containing selected terms based on Java program structure. We then classify the programs according to the implemented design patterns, and in this early work, three patterns are to be identified: Singleton, Composite and Observer. In order to search and retrieve source code from this digital library, similarity measurement is undertaken between users query and the index files. As the construction of an index file for both program submission and user query needs 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, relation between these sub-systems has to be identified. For example,

different result sets are generated if retrieval is undertaken based only on program template [20] or design pattern approach. The system is capable of achieving a better retrieval by cooperating these sub-systems using certain strategies. Therefore, to facilitate both processes (task decomposition and identification of cooperation strategies), we propose the use of agents in indexing and retrieving program source code.

3. Agent-based Architecture

Similar to the work done in RETSINA [18], we classify our agents into three types: *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).

User Interface Agent	Information Agent	Task Agent
PRA	RA	IBA
IRA	PA	ICA
QRA		PMA
RRA		STEMA

Table 1. Classification of Agents

The software digital library is based on the cooperation of the interface, task and information agents. In figure 1, we illustrate the general multi-agent system architecture for program indexing and querying in program repository. 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.

Each of the agents classified in table 1 is defined in detail below.

User Interface Agent (UIA) has two different roles in this architecture. From the programmers view, it is responsible for accepting programs or a project folder to be submitted to the program repository. The project folder contains a number of Java programs organised as a Java package. On the other hand, if a user wants to use it as a

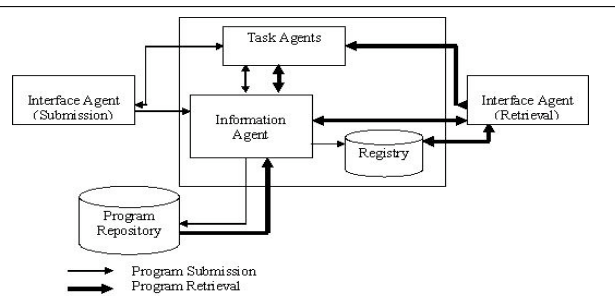


Figure 1. The Proposed Multiagent System Architecture for Program Indexing and Querying

search mechanism, the UIA will accept search queries. Each of the input messages will be given an ID to differentiate whether it is a program submission or only a search query. This ID is important to determine if the index to be generated should be stored in the registry. Users are allowed to submit two kinds of queries: phrase and program. 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) represent 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 to manage the index registry. All program index files (including programs full path name) are stored in this registry and these files are used in the matching process.

The Program Agent (PA) is another information agent used in this multiagent system. It manages the program repository by storing and retrieving the required Java files, for example retrieving the selected program source code to be presented to the users.

The Index Creation Agent (ICA) consist of three agents: Keyword agent (KEMA), Design Pattern agent (DEPA) and Java Template agent (TEMA).

- Similar to text mining, in KEMA, each word of the

Java program will be analyzed 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 - *the, a, an, void, and etc..* Upon removing these words, the selected words (after word stemming by STEMA) then be used as the metadata for the particular Java file. An index built based on the processed Java file will then consists of: (1) term, (2) type of the term, and (3) file name together with its full path.

Based on Query 2, three terms are extracted from the source code: *registry*, *addtoCount* and *getRegistry*. We classify these terms using Java program structure: class name, method name, package name or words that are in the comment section of a program. For example, statement *public class Registry* will produce term *registry* with *class* as the type of term. Therefore the index file for the program example query will contain the following:

```

projectRegistry.java
registry, class
addToCount, method
getRegistry, method
  
```

- As for TEMA, it is responsible for extracting information based on program structure [20]: (1) class name(s), (2) file name together with its full path, (3) method name and signatures, (4) superclass, (5) abstract class, (6) interface class. Example of indices generated by this agent are as following:

```

projectRegistry.java
method addToCount - parameters: null; return: null
getRegistry - parameters:null; return: registry
  
```

- DEPA is responsible for identifying three design pattern that are implemented in a Java program. It 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(percentage of design patterns existence). Generally, in our approach, a program is identified to be implementing Singleton if: (1) it only allows the creation of a single instance of a class; (2) the access to the private class variables is implemented in a public method. To identify the existence of Composite design pattern, DEPA: (1) identifies classes implementing at least one interface; (2) determines whether the identified classes provide method that receives interface class as its arguments.

In detecting Observer pattern, we have to identify the observers, the object to be observed and method(s) used to update any changes that to be made. For a class

to be identified implementing 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) class's constructor that receives at least one element from (1) as its method argument.

These three agents cooperate between each other in order to fulfill 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 request KEMA to identify lexical terms. With these terms, TEMA will than generate further indices for the Java template. DEPA is then invoked to determine the existence of any defined design patterns.

STEMA is an agent created with the purpose of stemming a word to its root form. In most cases, morphological variants of words have similar semantic interpretations and can be considered as equivalent for the purpose of IR applications. Thus, the key terms of a query or document are represented by stems rather than by the original words. For example, in Query 1, the root word of *implementing* is *implement*.

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) file name with its full path; (2) vector of terms object containing terms and type of the terms. Types of terms are determined based on program structure - class name, method name, package name and comments; (3) vector of class objects containing the class name, superclass name, method signatures, abstract class and a vector of interface class; (4) percentage of design patterns existence

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 Java file. This report is then presented to the person who submitted the program. Based on the above query example, the generated report contains the following:

```
File name :project1registry.java
Class name= registry
No.of selected terms=3
Design Pattern= Singleton(100%)
```

The Query Representation Agent (QRA) is responsible for formatting users query 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 methods: a description of what they are searching for in natural language and Java template of the query.

The Program Matcher Agent (PMA) is responsible for

finding all suitable Java programs. The similarity comparison is undertaken between two index files: query index file which is generated by the IBA, and index file for all programs stored in a registry. Indices in the query index file are mapped against all indices in the registry using 2 similarities measurement; string and design patterns. String and substring matching is undertaken based on the Levenshtein distance function [1]. This function is defined as the minimum number of characters, insertions, deletions and substitutions that need to be perform in any of the strings to make them equal. A threshold value is to be requested from the user in order for PMA to find similar Java programs that contain terms which produce the minimum value of the distance function. Design patterns similarities between users query and index file is obtained by evaluating the percentage value between these files. The selected Java files are then ranked according to their sum of distance function values and percentage of design patterns existence, where the program with the lowest and highest value, respectively, are presented as the most suitable program.

Upon having similarity between users index and the registry (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 containing all suitable Java programs. It is also responsible to retrieve 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 whoever 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, agent starts planning using its own operator and behaviour. If it requires other operator which does not exist in its domain knowledge, it must find and request other agent which 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 figure 2, we illustrate a portion of the plan library containing general descriptions of action decompositions methods written in an expression of 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}))

```

Figure 2. Action Descriptions for the Index Building of a Program

4. Agent Interaction for Program Submission and Program Retrieval

A user can either be a programmer/developer who wants to submit their Java application into the repository, or a user who wants to search and retrieve Java programs.

4.1. Program Submission

Based on the users program shown in Section 3, figure 4 illustrates the sequence diagram of agent communication during the process of program submission.

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, IA then informs IBA the current address of the project, or else it stores *Registry.java* in the repository and inform 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 ICA agents agree to perform the task, each of the ICAs request STEMA to perform stemming within the Java file. They will then inform IBA about the indices that they have created based on the evaluated *Registry.java* file. Upon receiving these indices, IBA builds 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 creates an index file con-

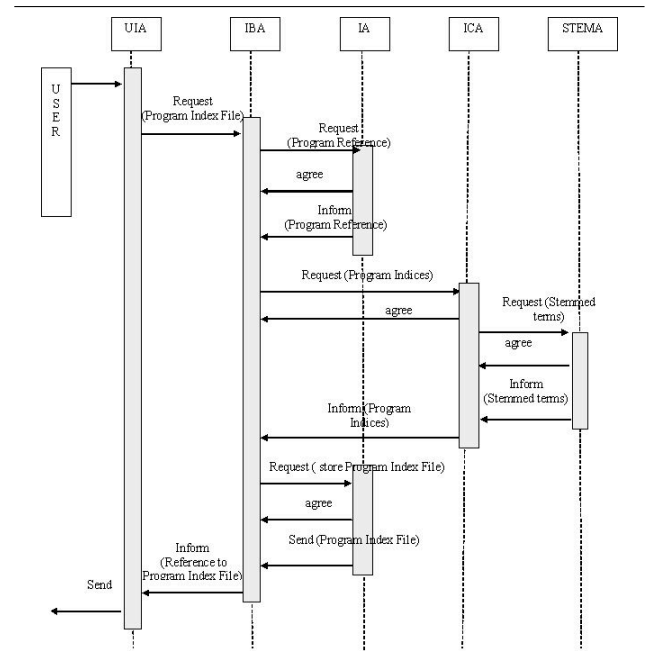


Figure 3. Sequence diagram for program submission

taining the data structure and stores it in the index registry. IBA then returns back to UIA, providing a Web reference for the generated index file.

4.2. Program Retrieval

Two different queries are currently supported: (i) keyword or phrase (natural language) describing users requirements and (ii) Java program or template. In 4, we describe how agents perform their task of retrieving relevant Java programs based on text phrase.

If a user submits a description of their requirements, *registry class implementing Singleton*, to the system, the UIA requests the IBA to build an index to represent this query. To perform this task, 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. ICA, then requests the IBA to build an index for the users' query (query index). Upon receiving this index from IBA, UIA then requests PMA to deliver Java programs that are relevant to the users query. To fulfill this task, PMA requests the information agent(RA) to sequentially retrieve index files stored in the program registry. It is then PMA's task to compare between these files (user query and program index files) and store the reference(s) for the matched

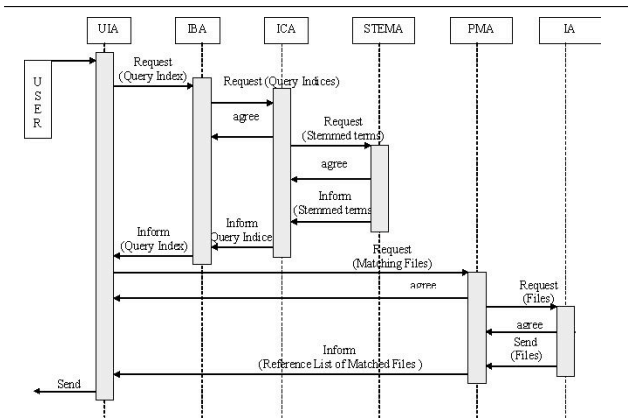


Figure 4. Sequence diagram for program retrieval

Java file(s). Upon completing the search, PMA informs UIA the list of relevant files.

On the other hand, if a user submits a Java program as his query to the system, 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 fulfill this task, IBA then request 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 request PMA to search for Java programs that are relevant to the search query, and this process continues as described in the above paragraph.

5. Case Study and Result Discussion

To determine the ground truth of detecting design patterns in source code digital libraries, we downloaded(from the Internet) 21 Java applications which are notified by the developers to have implement design patterns, namely, Singleton, Composite and Observer. These applications constitute of 67 Java files and in each design pattern group, 7 files are notified to implement the design patterns. Even though not all Java files implement design patterns, nevertheless, we may identify a file which implements more than one design pattern. We run the design pattern detector individually on each of the relevant design patterns group(i.e. Singleton query is posted on Singleton examples). Upon retrieving the result, recall analysis has been undertaken. A higher percentage of recall is obtained in retrieving Singleton files while both Composite and Observer produced 57.14% respectively. Based on the promising result, depicted in table 2, we then perform experiment using 7 applications(477

Design Patterns	No. of Relevant Files	No. of Retrieved Files	Recall
Singleton	7	5	71.43%
Composite	7	4	57.14%
Observer	7	4	57.14%

Table 2. Design Patterns in Java Applications

Design Patterns	Precision	Recall
Singleton	4%	100%
Composite	56%	37.84%
Observer	88%	81.48%

Table 3. Precision and Recall Analysis

files) representing mathematical domain, obtained from the SourceForge.net repository.

We summarize the traditional measures of retrieval performance, Recall (completeness of retrieval) and Precision (purity of retrieval) based on the first 50 ranked documents for all three design patterns in table 3. In general, the retrieval system has been able to retrieve 74 out of 130 relevant files which results 56.92% of success rate. Both detection of Composite and Observer design patterns produced more than 50% of precision while only 4% has been obtained for Singleton design patterns. We illustrate precision and recall graph for all three design patterns in figure 5.

Since there are only 2 relevant Java files which have been identified to implement Singleton design patterns, our system has managed to retrieved both of these files. The first relevant file is ranked first during the retrieval process while the second Singleton file was the last file to be retrieved(50th out of 50).

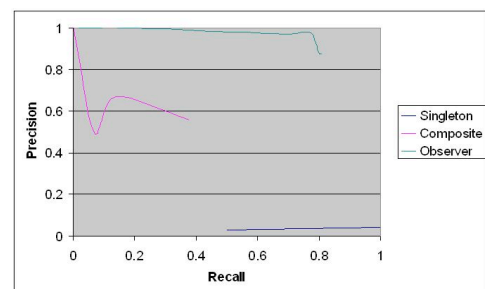


Figure 5. Precision vs. Recall

For Composite design pattern, the retrieval performance analysis started by obtaining value 1 and 0.014 for precision and recall. Precision is then slowly decreased before we retrieved the 10th document where it increased to the value of 0.67. However, by the end of the retrieval, recall value has raised to 0.378 which result a decrease in preci-

sion. Figure 5 also illustrates a balance trade-off between precision and recall as both analysis reached more than 80% of value at the end of the retrieval. Using 50 as the cut-point of retrieval, our multiagent system was capable of retrieving 81.48% of the identified relevant Java files.

Based on the capability of retrieving 74 out of 130 relevant files, we believed our retrieval system is capable to motivate developers in software reuse. As precision compensate recall, adjusting the threshold value would produce different sets of retrieved files and therefore generates different precision/recall measures.

6. Conclusion

With the emerging interest in making source code available, and the significant emphasis being placed on this by many software architects, DLs 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 utilising 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 source code digital library but also assist developers in identifying general design that addresses a recurring design problem in object-oriented systems. As most of the programmers and developers learn by studying available codes, being presented by various programs (which are relevant to the queries) is believed to motivate code and concept reuse.

References

- [1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, January 1999.
- [2] A. Barth, M. Breu, A. Endres, and A. de Kemp, editors. *Digital Libraries in Computer Science: The MeDoc Approach*. Springer-Verlag Heidelberg, 1998.
- [3] W. P. Birmingham, E. H. Durfee, T. Mullen, and M. P. Wellman. The distributed agent architecture of the university of michigan digital library (extended abstract). In *(AAAI) Spring Symposium on Information Gathering*, 1995.
- [4] D. Derbyshire, I. A. Ferguson, J. P. Muller, M. Pischel, and M. Wooldridge. Agent-based digital libraries: Driving the information economy. In *Proceedings of the Sixth IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 82–86, 1997.
- [5] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.
- [6] Y. Kusumura, Y. Hijikata, and S. Nishida. Text mining agent for net auction. In *ACM Symposium on Applied Computing*, pages 1095–1102, Nicosia, Cyprus, March 2004.
- [7] C. N. Linn. A multi-agent system for cooperative document indexing and query in distributed networked environments. In *Proceedings of the International Workshop on Parallel Processing*, pages 400–405, Japan, September 1999.
- [8] A. Mili, R. Mili, and R. T. Mittermeir. Storing and retrieving software components: A refinement based system. *IEEE Transactions on Software Engineering*, 23(7):445–460, July 1994.
- [9] S. Nakkrasae and P. Sophatsathit. A formal approach for specification and classification of software components. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 773–780. ACM Press, New York, 2002.
- [10] E. Ostertag, J. Hendler, R. P. Daz, and C. Braun. Computing similarity in a reuse system: An al-based approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(3):205–228, 1992.
- [11] J. Penix and P. Alexander. Using formal specifications for component retrieval and reuse. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, pages 356–365, 1998.
- [12] R. Prieto-Diaz. *A Software Classification Scheme*. Phd thesis, Department of Information and Computer Science, University of California, 1985.
- [13] H. Rodriguez. Good programming practice. http://www.start-linux.com/articles/article_75.php.
- [14] P. Ruben and F. Peter. Classifying software reuse. *IEEE Software*, 4(1):616, 1987.
- [15] P. Santanul and P. Atul. A framework for source code search using program patterns. *IEEE Transaction on Software Engineering*, 20(6):463–475, 1994.
- [16] J. Schumann and B. Fischer. Nora/hammr: Making deduction-based software component retrieval practical. In *Proceedings of the 1997 International Conference on Automated Software Engineering (ASE'97)*, pages 246–254, Lake Tahoe, CA, 1997.
- [17] V. Sugumaran and V. C. Storey. A semantic-based approach to component retrieval. *The DATA BASE for Advances in Information Systems*, 34(3):8–24, Summer 2003 2003.
- [18] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert*, 11(6):36–46, December 1996.
- [19] S. Ugurel, R. Krovetz, and C. L. Giles. 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*, pages 632–638. ACM Press, 2002.
- [20] Y. Yusof and O. F. Rana. Template mining in source code digital libraries. In *Proceedings of the International Workshop on Mining Software Repositories, 26th International Conference on Software Engineering*, pages 122–126, Edinburgh, UK, 2004.

Architecture for Distributed Agent-Based Workflows

C. Reese, J. Ortmann, S. Offermann, D. Moldt, K. Lehmann, T. Carl
Computer Science Department
University of Hamburg

{reese, ortmann, offerma, moldt, lehmann, carl}@informatik.uni-hamburg.de

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.

A further application area of workflow management systems 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 demand. A typical example for such a setting, where a workflow needs to be constantly adopted are virtual enterprises, where changing partnerships lead to changing requirements.

Based on the formal modeling 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

To build distributed applications different concepts and technological means are used. New areas like interorganizational cooperations and virtual enterprises require new solutions due to the high dynamics in their interrelations. Since the process perspective has been within the center 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. (see [13]).

The main contribution of this work is our approach of fragmenting workflows for distributed execution with supporting protocols and architecture. This architecture is of particular strength due to its agent orientation, its formal basis provided by Petri nets and its partial tool support.

An implementation by Carl [7] allows the splitting of workflows into arbitrary fragments. These fragments, encapsulated by agents, are treated again as workflows and they can be executed at different locations using different workflow enactment systems, which are the conceptual platforms of the agents. Therefore, we provide a concept for a distributed and concurrent workflow management system, 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. Fragmentation of 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, Figure 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 [24]). These agents provide the functionality of distributed agent based workflows to any application. In the following, the layers are described in more detail.

2.1. Reference Nets and RENEW

For an introduction to reference nets, see [16]. Reference nets (published first in [17]) are an extension of the Coloured Petri net (CPN) formalism (for extensive introduction, see [14]) adding both the concept of nets-within-

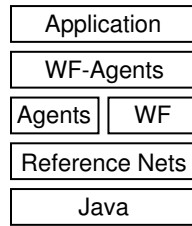


Figure 1. Simple Architecture Overview

nets introduced by Valk in [23] and the concept of synchronous channels (as first introduced in [8]). 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 [22]), 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 expressive power of Petri nets, this makes reference nets a good choice to model and execute workflows.

2.2. Workflow Nets

The use of Petri nets in the workflow area has been thoroughly investigated (see [1]). Workflow patterns can be expressed by Petri nets (see [2]). Reference nets are especially suitable for defining a workflow due to their high expressional 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 (see [11]) is used 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 [3] and on the persistent Petri net execution engine presented in [12].

2.3. Agents

The technical agent framework CAPA (Concurrent Agent Platform Architecture, see [9]) 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 [10]).

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 like sending and receiving messages. The behavior 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 in [6].

2.4. The Components of a WFMS

The structure model for workflow systems of the Workflow Management Coalition (WfMC, [25]) defines six basic components of a WFMS, not repeated here (see [24]). 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 system.

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 other 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 the other components concerning running and finished tasks or problems. This provides a

view to the system state as far as possible within distributed systems.

2.5. Relation between Workflows and Agents

In our architecture, workflows are encapsulated by agents. Agents may migrate to other platforms. This solves the distribution of workflows and workflow fragments. The agent framework used here provides agent mobility (see [15]). Agents are encapsulated components that can be accessed only via their communication interface. Access to the agent-internal workflow therefore must be explicitly allowed by the agent (this does not take into account the general problems and challenges in the agent security area, which are not discussed here).

Beside technical advantages, the conceptual advantages of Petri nets and agents are combined as well. 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.

Since the protocol nets we use to specify the behavior of agents already have a workflow-like structure, we use Petri nets both for agents and for workflows. This restricts the architecture of the WFMS. Reference net models for a specific application are restricted to be workflow conform and agent conform at the same time.

2.6. Further Procedure

The overall way to our solution is now: First, we enable the definition of cut-off points within a workflow specification and with that we enable the distribution of workflow fragments. Second, 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 neighbored elements need to be examined and synchronized. The requirements for a workflow fragmentation are:

- (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 [2].
- (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 to other actions. The border transition would be a coercion 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 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 can not 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 Figure 2: 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).

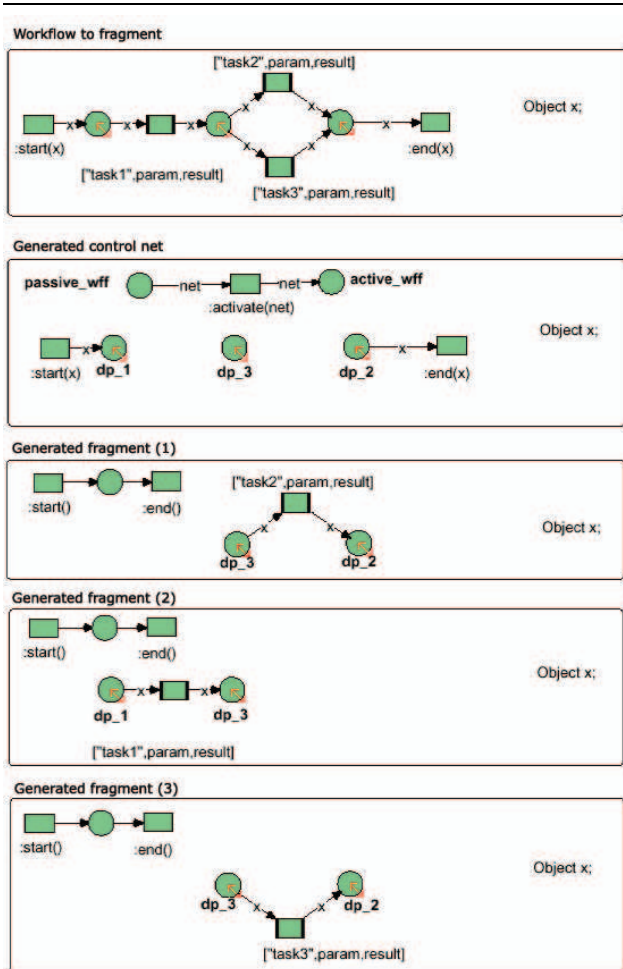


Figure 2. Example workflow fragmentation

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. 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. 2, 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. Gather the names of connected nodes for each fragment (This results in five fragments, one of them containing node `start` and node `dp_1`).

- (5) When no unvisited 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 two 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).

Result: An error message if the net is inconsistent, otherwise disjunct 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 or test 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 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. A garbage collector removes the net instance from the memory as soon as the space is needed otherwise. 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 is activated the first time one of its border places gets a token and it is deactivated only when the `end` transition in the control net is activated.

Workflow nets should therefore be designed in such a way that nothing happens once the `end` transition was activated. Beside others, 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 dis-

tributed workflows [18]. Other approaches were motivated by load-balancing issues as in [4], such that workflows can be redistributed to other servers according to their load. The coordination of Web services is addressed in [5].

Furthermore, we focussed on the development of a FIPA-compliant framework, that 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 modeling of the system on the other hand. reference nets are used for the modeling of the system as well as for the modeling 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 [20], 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.

The following sections describe our design of workflow agents building upon agent and workflow technology as depicted in Figure 1, and its integration into existing components.

4.1. Plug-in Dependencies

The dependencies of the different plug-ins are shown in Figure 3. 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, 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 (compare Figure 4).

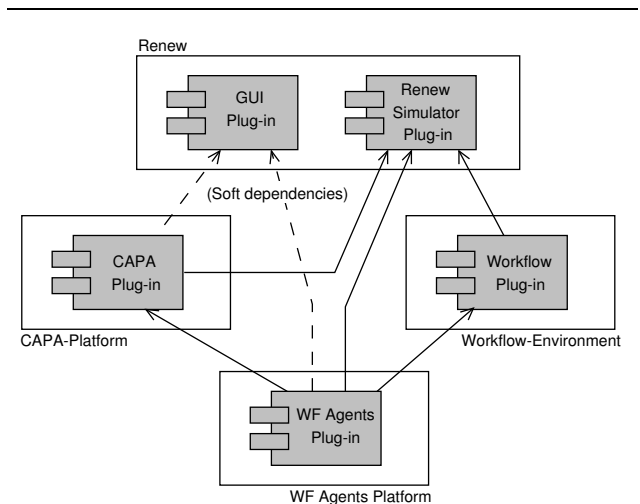


Figure 3. Dependencies among RENEW plug-ins

WFES agent The *Workflow Enactment Service* agent forms the platform of the WFMS containing all workflow specific agents except the application specific task agents. The WFES agent manages the system and forms the interface to other WFMS.

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-CI agent Within the *Workflow Client* agent the users of the system are managed and communicated with. A participating user registers using this interface and is assigned to services he offers or uses according to his role. This is the workflow client application in the classical sense.

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 on exceptional situations.

Task agent Task agents correspond to the “invoked application” in the WfMC model. They are arbitrary agents which can be provided by an application. Their supplied services are called by a task if this is required by a workflow. This agent type is not contained in the platform provided by the WFES agent.

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.

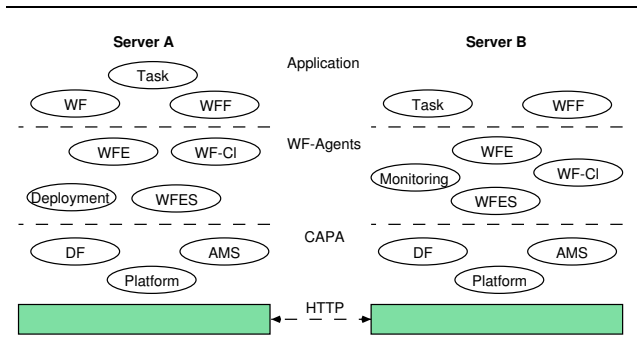


Figure 4. Example infrastructure of WF platforms

Figure 4 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 the platform itself which is implemented as an agent in CAPA. Again above are the agents of the WF agent platform. The *Workflow Enactment Service* 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.

4.3. The Running System

In the following, some aspects of the running system are discussed.

4.3.1. Distribution and Execution After the fragmentation, each fragment is encapsulated within one WFF (*Workflow Fragment*) agent. These are transferred to their execu-

tion 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 (see Section 4.3.3), 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 that 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.

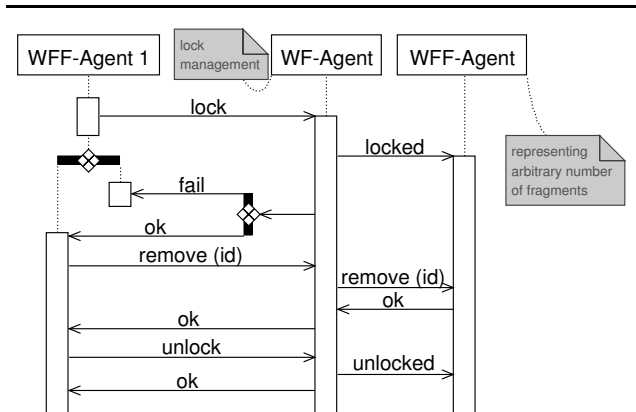


Figure 5. Synchronization protocol for a border place

The coordination and conflict solving is done by the WF agent which also manages the lock. The resulting topology has star shape, so the fragments do not need to know each other. Figure 5 illustrates this. Further details are described in [7].

Starting and ending workflows or parts of workflows happens through message exchange: a workflow can not 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.

4.3.2. 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 WFES (workflow enactment service) 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.

4.3.3. 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 [21]), agents can search and publish services worldwide, e.g. within the open agent network openNet (see [19]).

5. 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 modeling 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 flex-

ible, dynamic and autonomous configuration of each workflow and platform. Since workflows are encapsulated by agents, these advantages can be transferred. 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 fulfill 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 modeling formalisms). Here the use of reference nets and a tool set which really supports such concurrency is of high value. To our knowledge there are no other available frameworks that cover concurrency and practical usability at the same time to the same extend on both levels, conceptually and technically.

Outlook There are several possibilities to extend our work so far. The current implementation is still not as elaborated as the conceptual parts are. Here the workload to really meet the technological / practical requirements has to be considered. Which parts should be extended will be driven by the development of a distributed software development environment. 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). Here agent and multi-agent system concepts will play a central role since the flexibility, openness, autonomy and mobility will become more and more important.

References

- [1] W. v. d. Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, number 1248 in LNCS, pages 407–426, Berlin, 1997. Springer.
- [2] W. v. d. Aalst, A. t. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
- [3] W. v. d. Aalst, D. Moldt, R. Valk, and F. Wienberg. Enacting Interorganizational Workflows Using Nets in Nets. In *Proceedings of the 1999 Workflow Management Conference*, volume 70, pages 117–136. University of Münster, 1999.
- [4] T. Bauer, M. Reichert, and P. Dadam. Intra-subnet load balancing in distributed workflow management systems. *Int. J. Cooperative Inf. Syst.*, 12(3):295–324, 2003.
- [5] M. B. Blake and H. Goma. Object-oriented modeling approaches to agent-based workflow services. In C. J. P. de Lucena, A. F. Garcia, A. B. Romanovsky, J. Castro, and P. S. C. Alencar, editors, *SELMAS*, volume 2940 of LNCS, pages 111–128. Springer, 2003.

- [6] L. Cabac, D. Moldt, and H. Rölke. A Proposal for Structuring Petri Net-Based Agent Interaction Protocols. In W. van der Aalst and E. Best, editors, *24th ICATPN 2003, Eindhoven, NL*, volume 2679, pages 102 – 120, Berlin, 2003. Springer.
- [7] T. Carl. Entwicklung eines agentenbasierten verteilten Workflow-Management-Systems mit Referenznetzen. Diploma thesis, University of Hamburg, 2004.
- [8] S. Christensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous communication. Technical Report DAIMI PB-390, Computer Science Department, Aarhus University, Apr. 1992.
- [9] M. Duvigneau, D. Moldt, and H. Rölke. Concurrent architecture for a multi-agent platform. In F. Giunchiglia, J. Odell, and G. Wei, editors, *AOSE 2002, Revised Papers and Invited Contributions*, volume 2585 of *LNCS*, Berlin, 2003. Springer.
- [10] Foundation for Intelligent Physical Agents. *FIPA Agent Management Specification*, 2004. <http://www.fipa.org/specs/fipa00023/>.
- [11] T. Jacob. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma thesis, University of Hamburg, 2002.
- [12] T. Jacob, O. Kummer, and D. Moldt. Persistent Petri Net Execution. *Petri Net Newsletter*, 61:18–26, Oct. 2001.
- [13] N. R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2):277–296, 2000.
- [14] K. Jensen. *Coloured Petri Nets: Volume I; Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.
- [15] M. Köhler, D. Moldt, and H. Rölke. Modelling mobility and mobile agents using nets within nets. In W. van der Aalst and E. Best, editors, *24th ICATPN*, volume 2679 of *LNCS*, pages 121–139. Springer, 2003.
- [16] O. Kummer. Introduction to Petri nets and reference nets. *Sozionik Aktuell*, 1:1–9, 2001. ISSN 1617-2477.
- [17] O. Kummer. *Referenznetze*. Logos, Berlin, 2002.
- [18] K. Lehmann and V. Markwardt. Proposal of an Agent-based System for Distributed Software Development. In D. Moldt, editor, *Proc of MOCA 2004*, pages 65–70, Aarhus, Denmark, Oct. 2004.
- [19] OpenNet project. <http://www.x-opennet.org/>, 2004.
- [20] M. Purvis, B. T. R. Savarimuthu, and M. K. Purvis. Evaluation of a multi-agent based workflow management system modeled using coloured petri nets. In M. Barley and N. K. Kasabov, editors, *PRIMA*, volume 3371 of *LNCS*, pages 206–216. Springer, 2004.
- [21] C. Reese, M. Duvigneau, M. Köhler, D. Moldt, and H. Rölke. Agent-based settler game. In *Agentcities Agent Technology Competition, Barcelona, Spain*, Feb. 2003.
- [22] RENEW – the reference net workshop homepage. URL <http://www.renew.de/>.
- [23] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In J. Desel, editor, *19th ICATPN*, number 1420 in *LNCS*, Berlin, 1998. Springer.
- [24] Workflow Management Coalition. WfMC workflow reference model. URL <http://www.wfmc.org/standards/model.htm>, 2005.
- [25] Workflow Management Coalition (WfMC). URL <http://www.wfmc.de/>, 2005.

OWL-P: A Methodology for Business Process Development*

Nirmit Desai Ashok U. Mallya Amit K. Chopra Munindar P. Singh
{nvdesai, aumallya, akchopra, singh}@ncsu.edu
Department of Computer Science
North Carolina State University

Abstract

Business process modeling 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 modeling and enactment which is centered 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 modeled 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. Further, conventional models do not facilitate flexible actions by the participants.

This paper proposes an approach for business process modeling 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 mul-

iple roles and address specific purposes such as ordering, payment, shipping, and so on. Protocols are given a 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 which one to perform. Typically, policies are business logic to generate 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) [11]. 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 [9]. 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. Shortcomings of Traditional Approaches

Consider a supply chain business process where a Customer, a Merchant, a Shipper, and a Payment Gateway collaborate to fulfill their business goals. Such a process can be captured via a traditional flow-based approach such as BPEL [3]. Such a representation would be functionally correct, but inadequate from the perspectives of open environments. The following are its shortcomings:

Lack of Semantics. Traditional approaches expose low-level interfaces, e.g., via WSDL [17], but associate no semantics with the participants' actions. This

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

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 SOC 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

Figure 1 shows our conceptual model for a 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. They correspond to service specifications in SOC terminology. 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 behavior of the respective role in the given 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 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

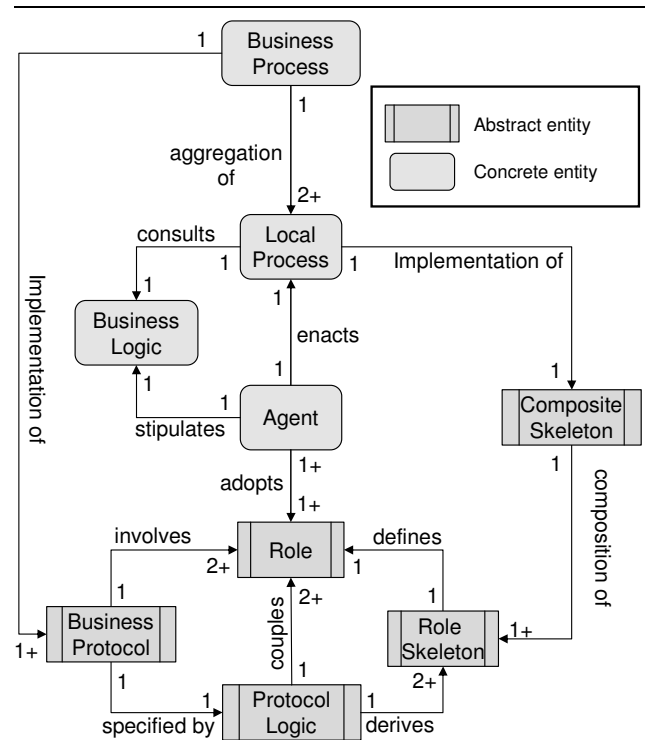


Figure 1. Conceptual model

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.

The concept of *commitments* has been proposed to capture a variety of contractual relationships, while allowing manipulations such as delegation and assignment, which are essential for open systems [14]. For example, a customer's agreement to pay the price for the item after it is delivered is a commitment that the customer has towards the merchant. Violations of commitments can be detected; in some important circumstances, violators can be penalized.

Definition 1 A commitment $C(x, y, p)$ denotes that the agent x is responsible to the agent y for bringing about the condition p .

Commitments can also be *conditional*, denoted by $CC(x, y, p, q)$, meaning that x is committed to y to

bring about q if p holds where, q is called the precondition of the commitment. The following are some of the conventional operations defined on commitments [14] employed in this paper.

1. $\text{CREATE}(x, c)$ establishes the commitment c in the system. This can only be performed by c 's debtor x .
2. $\text{CANCEL}(x, c)$ cancels the commitment c . This can only be performed by c 's debtor x . Generally, cancellation is compensated by making another commitment.

3. Protocol Specification

A business protocol is a specification of the allowed interactions among two or more participant roles. The specification focuses 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) [8] for defining rules. SWRL allows us to specify implication rules over entities defined as OWL-P instances. The availability of tools such as Protégé [12] is a motivation for grounding our approach in OWL.

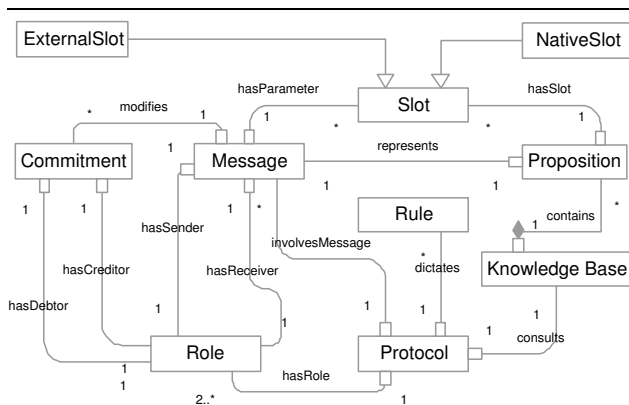


Figure 2. Basic OWL-P ontology

The important OWL-P elements and their properties are shown in Figure 2. 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. 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.

Figure 3 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 is the rule for sending quote message in the Order protocol in Figure 3:

```
contains(KB, reqForQuoteProp(?itemID)) =>
send(M, quote(?itemID, ?itemPrice)) ^
createCommitment(M, CC(M,C,pay(?itemPrice),goods(?itemID)))
```

Every message msg is represented in the knowledge base as msgProp . These protocol rules are abstract and they need to be augmented with the business logic to assign values to the message contents. OWL-P dictates that the rules having undefined native slots must be augmented with the business logic that produces such values. Operational semantics of the predicates are given in the next section. The OWL-P ontology and protocol instance examples in their RDF/XML serialization, and corresponding Protégé projects are available at <http://www4.ncsu.edu/~nvdesai/owl/>.

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 semantics 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

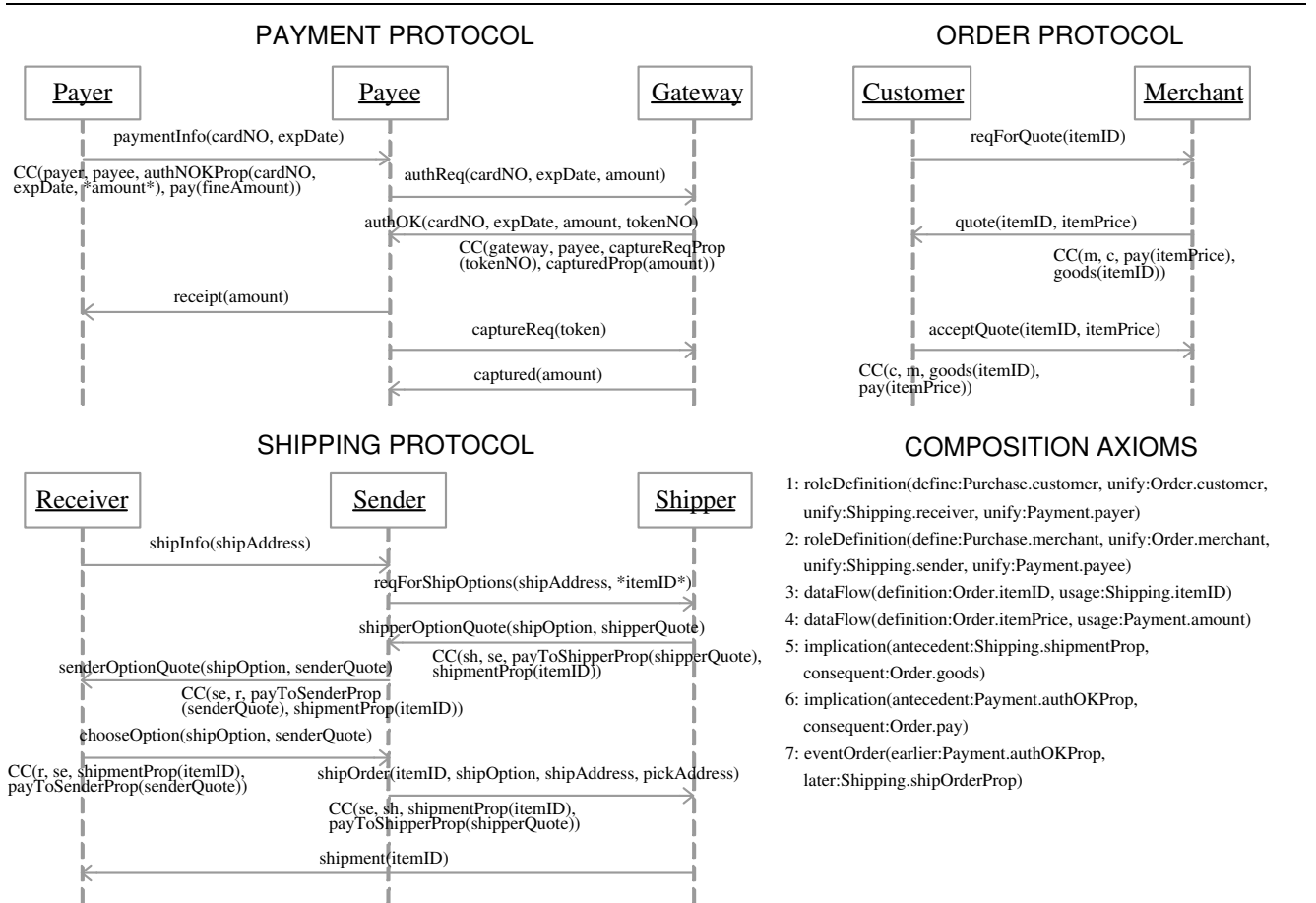


Figure 3. Example: Order, Shipping, and Payment protocols and their composition

Predicate	Domain	Range	Context	Meaning
contains	KnowledgeBase	Proposition	Body	Proposition \in KnowledgeBase ?
assert	Proposition	KnowledgeBase	Head	KnowledgeBase \leftarrow KnowledgeBase \cup Proposition
send	Role	Message	Head	Asynchronous send to the receiver assert(KnowledgeBase, MessageProp)
receive	Role	Message	Head	Asynchronous receive from the sender assert(KnowledgeBase, MessageProp)
createCommitment	Role	Commitment	Head	assert(Knowledgebase, CommitmentProp)

Table 1. Operational semantics of protocol rules

when the meaning is clear. Figure 4 shows an inside view of an agent to demonstrate how the rules govern the interactions.

4. Composite Protocols

The previous section described how to specify individual protocols. To meet the requirements of business pro-

cesses, 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

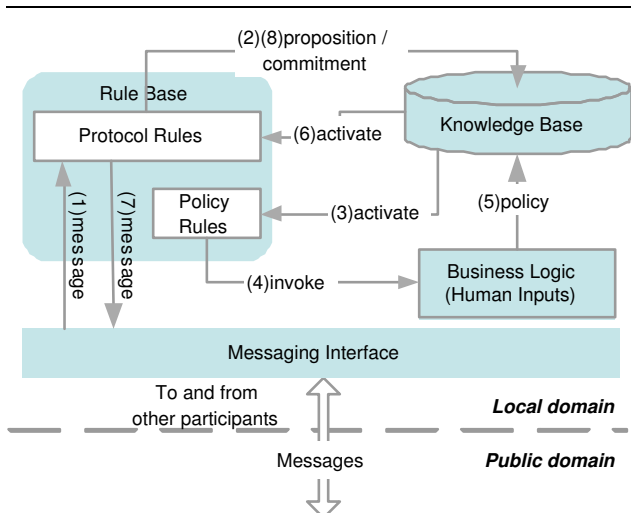


Figure 4. Agent architecture: protocol and policy interplay

construct such composite protocols? How do they facilitate reusability? How do they allow refinements of protocols? This section answers these questions.

4.1. Construction of Composite Protocols

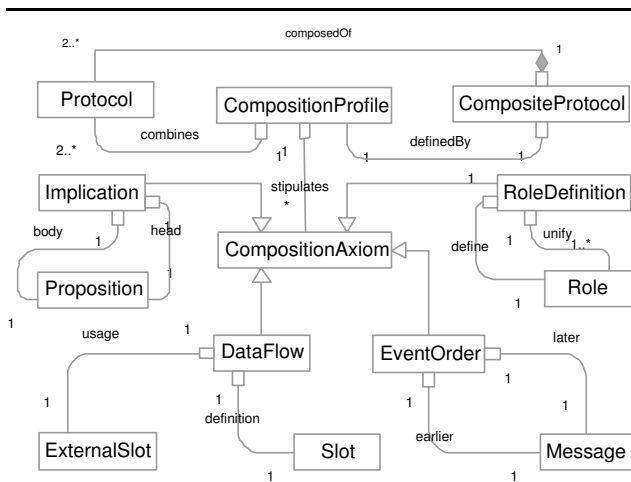


Figure 5. OWL-P composition classes and properties

Figure 5 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

define relationships among the protocols being combined. The operational semantics of an axiom specifies the way in which the relationships affect the composite protocol. Figure 3 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 played 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 [7]. 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 subclasses 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.

Definition 2 *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 Figure 3, the Order protocol is open as its rules do not assert propositions `pay` and `goods` necessary for discharging the commitments created. The Payment, 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 $C(m, c, \text{goods}(\text{itemID}))$ 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. A more flexible solution would be to allow the merchant to refund money and release it from the commitment, provided the customer agrees to it. We can achieve this flexibility by combining the purchase protocol with an adjustment protocol as discussed in [7].

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 [7]. 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 Figure 3, 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

$$\text{shipperOptionQuoteProp}(\dots) \Rightarrow \text{senderOptionQuote}(\dots) \wedge \text{CC}(\text{Se}, \text{Re}, \text{payToSenderProp}(\dots), \text{shipmentProp}(\dots))$$

Receiver Skeleton Rule

$$\text{shipInfoProp}(\text{?shipAddress}) \Rightarrow \text{receive}(\text{senderOptionQuote}(\dots)) \wedge \text{CC}(\text{Se}, \text{Re}, \text{payToSenderProp}(\dots), \text{shipmentProp}(\dots))$$

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, and 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.

$$\text{startProp} \Rightarrow \text{receive}(\text{C}, \text{reqForQuote}(\text{?itemID}))$$

$$\text{reqForQuoteProp}(\text{?itemID}) \wedge \text{quotePolicy}(\text{?itemPrice}) \Rightarrow \text{quote}(\text{?itemID}, \text{?itemPrice}) \wedge \text{CC}(\text{M}, \text{C}, \text{pay}(\text{?itemPrice}), \text{goods}(\text{?itemID}))$$

$$\text{quoteProp}(\text{?itemID}, \text{?itemPrice}) \Rightarrow \text{receive}(\text{C}, \text{acceptQuote}(\text{?itemID}, \text{?itemPrice})) \wedge$$

CC(C, M, goods(?itemID), pay(?itemPrice))

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 merchant agent is shown above. The last rule is the policy rule which 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

Figure 6 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 Figure 4 whose details are given in [7]. 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 [3] 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 [6], which includes a process model for Web services uses semantic annotations to facilitate dynamic composition. A composed service is pro-

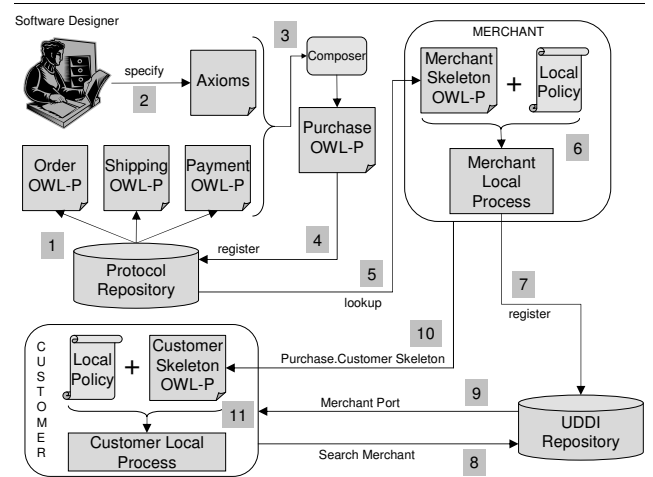


Figure 6. Usage scenario

duced 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. Our goal is to provide a human designer with tools to facilitate service composition. Unlike BPEL, which specifies the internal orchestration of services, WSCI [16] specifies the conversational behavior 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.* [15] employ interval temporal logic to specify and verify ongoing behavior of a composed service. Their use of “assumption” and “commitment” (different meaning than here) assertions allows better compositionality. Gerede *et al.* [5] treat services as activity-based finite automata to study the decidability of compositionality and existence of a lookahead 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 [9] treat these concepts formally. The MIT Process Handbook [10], 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 similarity with the Hand-

book. RosettaNet [13] is similar to our approach in that it centers 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 [2]. Tropos [4] differs from these in that it includes an early requirements stage in the process. Gaia [18] differs from others in that it describes roles in the software system being developed and identifies processes that they are involved in 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.* [1] 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 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 have been defined [13], this approach will enable 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 interesting 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.

References

- [1] K. Baïna, B. Benatallah, F. Casati, and F. Toumani. Model-driven web service development. In *Proceedings of Advanced Information Systems Engineering: 16th International Conference, CAiSE*, June 2004.
- [2] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, editors. *Methodologies and Software Engineering for Agent Systems*. Kluwer, 2004.
- [3] BPEL. Business process execution language for web services, version 1.1, May 2003. www-106.ibm.com/developerworks/webservices/library/ws-bpel.
- [4] P. Bresciani, A. Perini, P. Giorgini, F. Guinchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.
- [5] Çağdaş Evren Gerece, R. Hull, O. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proceedings of the International Conference on Service Oriented Computing*, 2004.
- [6] DAML-S. DAML-S: Web service description for the semantic Web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, July 2002. Authored by the DAML Services Coalition, which consists of (alphabetically) Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew McDermott, Sheila A. McIlraith, Srinu Narayanan, Massimo Paolucci, Terry R. Payne and Katia Sycara.
- [7] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh. Processes = protocols + policies, a methodology for business process development. Technical report, Department of Computer Science. North Carolina State University, 2004. TR2004-34.
- [8] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosz, and M. Dean. SWRL: A semantic web rule language combining OWL and RuleML, May, 2004 (W3C Submission). <http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/>.
- [9] A. U. Mallya and M. P. Singh. A semantic approach for designing commitment protocols. In *Proceedings of the AAMAS-04 Workshop on Agent Communication*, July 2004. To appear.
- [10] T. W. Malone, K. Crowston, and G. A. Herman, editors. *Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA, 2003.
- [11] OWL. Web ontology language, Feb 2004. <http://www.w3.org/TR/2004/REC-owl-features-20040210/>.
- [12] Protégé. The protégé ontology editor and knowledge acquisition system, 2004. <http://protege.stanford.edu/>.
- [13] RosettaNet. Home page, 1998. www.rosettanet.org.
- [14] M. P. Singh. An ontology for commitments in multiagent systems: Toward a unification of normative concepts. *Artificial Intelligence and Law*, 7:97–113, 1999.
- [15] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. In *Proceedings of the International World Wide Web Conference*, pages 544–552, 2004.
- [16] WSCI. Web service choreography interface 1.0, July 2002. www.sun.com/software/xml/developers/wsci/wsci-spec-10.pdf.
- [17] WSDL. Web Services Description Language, 2002. <http://www.w3.org/TR/wSDL>.
- [18] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The gaia methodology. *ACM Transactions on Software Engineering Methodology*, 12(3):317–370, 2003.

An Ontology Support for Semantic Aware Agents

Michele Tomaiuolo, Paola Turci, Federico Bergenti, 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

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. In order to autonomously achieve these strategic and ambitious objectives, agents should be enhanced with suitable tools and mechanisms.

In this paper we concentrate on what we consider the central issue when moving towards the vision of semantic multi-agent systems: the management and exploitation of OWL ontologies. In particular, we present a two-level approach, which copes with both the issues of managing complex ontologies and of providing ontology management support to lightweight agents. The key feature that distinguishes our approach from others is the fact that a light ontology support is embedded in each agent whereas one or more dedicated agents, acting as ontology servers, provide a more expressive and powerful ontology support to the agents that need it.

1 Introduction

The work presented in this paper is an attempt to bridge two co-existing realities: 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, enabling 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 focus on what we consider the central theme when moving towards the vision of semantic multi-agent systems: an 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 not to be feasible. In our opinion, a good compromise is represented by a two-level approach: a light ontology support embedded in each agent and one or more dedicated agents, called ontology servers, providing a more expressive and powerful

ontology support to the agents of the systems

In the next section we examine the rationale of our choice of embedding a light ontology support in each agent of a multi-agent system. Agents refer to this ontology support when expressing the content of ACL messages, e.g., when expressing the concepts of the domain and the relationships that hold among them. Section 3 describes the library that we have realized to provide agents with the aforementioned two-level ontology management support. 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 we situate our research is characterized by different domain knowledge modelling techniques and by different needs. On one hand there is the semantic Web and OWL [13], 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 of having an ontology representation more in line with the object-oriented data 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 the Java technology.

At this point a crucial question arises: is the semantics implied by the object-oriented paradigm powerful enough? A comparison between the two models (object oriented data model, i.e., 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 in some specific cases. 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 data model, i.e., without considering the

possibility of defining a meta-model, what we are able to express is a taxonomy among classes¹.

Briefly, we can rephrase the object-oriented data 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 objects². 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 *SHIQ* Description Logics. OWL benefits from years of DL research and it can rely on a well defined semantics, known reasoning algorithms and highly optimised implemented reasoners.

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 they 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 data model. Indeed, they are both based on the notion of class: in the object oriented data 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 notice that the two notions are fairly different. Formally [5], considering an interpretation I that consist of a set \mathcal{A}^I (the domain of the interpretation) not empty and an interpretation function I , to every atomic concept A is assigned a set $A^I \subseteq \mathcal{A}^I$ and to every atomic role R a binary relation $R \subseteq \mathcal{A}^I \times \mathcal{A}^I$. 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. $R^I \subseteq \mathcal{S}^I$ 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 [11], 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 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 data 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 knowledge base of agents. 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 several research work was devoted to deal with the comparison between OWL and UML [1,6]. Among these, some considered the mapping related to a particular object-oriented programming language: Java. Focusing 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 [4,9] and OWL API [3,12]. The latter consists of a high-level programmatic interface for accessing and manipulating OWL ontologies. Its aim is implementing 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

¹ We focus on the semantics of the so called "class-based" data model.

² The dynamic properties of the model are not dealt with in this paper, focused on the structural aspects, even if they constitute an important part of the model.

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 it lacks 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 is the purpose of the mapping. For example, the aim of having a full mapping and preserving the semantics is too strong in our case. We can relax this constraint and we can be satisfied with a partial mapping. This partial mapping is required only to be consistent in the sense that it does not preserve semantics but only semantic equivalence [2]. 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 let us use, e.g., renaming and redundancy in order to achieve this goal, like in the use of interfaces in Java in order to express the 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 Java class for each ontology class is necessary to overcome the single-inheritance limitation that applies to Java classes³. Each interface, instead, can extend an arbitrary number of parent interfaces. The corresponding class is eventually obliged to provide an implementation of all accessor methods defined by each of the directly and indirectly implemented interfaces.

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 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 this and other similar properties (e.g. *equivalentClass*), that is the representation of the world that we can state using this model can only refer to a “closed-world” assumption. This constitutes a limit when the knowledge representation is applied in situations where one cannot assume that the knowledge in the knowledge base is complete.

As far as properties are concerned, since they are not first-class modelling elements in Java, it is not possible to create property hierarchies, 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 variables whose type is the class specified in the property’s range. There are a number of special constraints that it is possible to enforce on properties:

1. Cardinality constraints state the minimum and maximum number of objects that can be related;
2. Domain limits the individuals to which the properties can be applied;
3. Range limits the individuals that the property may have as its value.

Accessor methods could ensure that cardinality constraints be satisfied, but this information is implicit and embedded in the source code of the class 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 domain of a property is specified as a single class, the corresponding Java interface contains declarations of accessor methods for the property. In the case of a

³ The Java generalization involves also the behavioural aspects and so it is semantically different from the OWL *subClassOf* property. This mapping nevertheless preserves semantic equivalence.

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 alternate domains are defined using the *unionOf* operator; we can cope with this creating a union interface but with the limitations expressed above.

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.

From the previous analysis it emerges clearly that the expressiveness of the Java language is lower even to OWL Lite, but despite this in our view it is still valuable with respect to the agent needs.

2.2 Reasoning about Knowledge

Although DLs (and hence OWL DL) and object-oriented data models have a common root, class-based data 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 explicitly 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 quite common complex reasoning service, i.e. classification, also plays a marginal role in an object oriented data 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 which differentiates more 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 data model exhibits a limited use of entailment. Inheritance may represent a simple way of express implicit knowledge (class inherits all the properties of its parents without explicitly 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 to

store 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 being in a concept expression. The information retrieval task plays a leading role in a knowledge base centered 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. In particular, the proposed two-level approach to ontology management in multi-agent systems is implemented as a toolkit providing the following functionality:

1. Import OWL ontologies as an object-oriented hierarchy of classes;
2. Implement an Ontology Server to provide the centralized management of shared ontologies.

3.1 OWLBeans

The OWLBeans toolkit, 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 toolkit 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 toolkit, i.e., mainly JavaBeans and simple metadata representations used by JADE [7], are not able to express all the relationships that are present in the source. 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.

The main functionality of the presented toolkit is to extract a subset of the relations expressed in an OWL document for generating a hierarchy of JavaBeans reflecting them, and possibly for creating a corresponding JADE ontology to represent metadata. Anyway, given its modular architecture, it also allows 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.

3.1.1 **Intermediate ontology model.** The main objective of the OWLBeans toolkit is to extract JavaBeans from an OWL ontology. In order to keep the code maintainable and modular, we decided to create first 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 goals we fixed to design 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 to 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 in the whole design 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, as 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 even classes defined in external ontologies.

One of the main issues regarded properties, as they are handled in different ways in description logics and in object oriented systems, as described in details in the previous sections. While they are first level entities in Semantic Web languages, they are more strictly related to their “owner” class in the latter model. 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 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 toolkit is made of just few, very simple classes. The simple UML class diagram shown in Figure 1 describes the whole 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 intermediate model – references, ontologies, classes and properties – are implicitly *OwlResource* objects.

OwlReference is used as a simple reference, to point to super-classes and range types, and do not add anything to the *OwlResource* class definition. It is defined to underline the fact that classes cannot be used directly as ranges or parents.

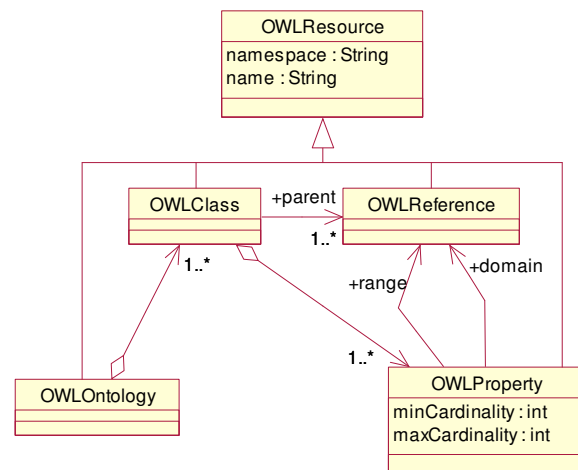


Figure 1 – Class diagram of the intermediate model

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 used to store respectively the minimum and maximum allowed cardinality for the property values. Moreover, 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 in some places. i.e., to point to super-classes and to allowed ranges. This decision has two main advantages over direct Java references to final objects: parsing an OWL file is a bit simpler, as references can point to classes that are not yet defined, and above all in this way super-classes and ranges are not forced to be local classes, but can be references to resources defined somewhere else.

In our toolkit, 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 the toolkit provides readers to inspect OWL files and JADE ontologies, and writers to generate OWL files, source files of JavaBeans and JADE ontologies.

3.1.2 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 very restrictive limitations, mainly caused by the rather strong differences between the OWL data model and the object-oriented data model. In fact, only few, basic features of the OWL language are currently supported.

Basically, the OWL ontology is first read into a Jena *OntModel* object and then all *classes* are analyzed. 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, which is <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 data model.

3.1.3 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 analyzed 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 [13], 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 toolkit 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. Each interface, instead, can extend an arbitrary number of parent interfaces. The corresponding class is eventually obliged to provide an implementation for all the methods defined by one of the directly or indirectly implemented interfaces.

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.

3.1.4 Additional components. Additional components are provided to read and write ontologies in different formats.

For example, the *JadeReader* class allows to load a JADE ontology, to save it in OWL format or to generate 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 to convert an ontology from its intermediate representation to an

OWL model. This is quite straightforward, as 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 toolkit, 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 they 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 to optionally use 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, as in this case the leading class name would be automatically stripped off by the *OwlFileReader* class.

3.1.5 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 [1], 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* [8], a Java scripting engine, into the toolkit. 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. But, 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, e.g., or provided to the application in other ways.

3.2 Ontology Server

The OWLBeans toolkit allows agents to import taxonomies and classifications from OWL ontologies, in the form of a hierarchy of Java classes. Anyway, 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, though 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 be added and removed at runtime, too.

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.

Anyway, for the Ontology Server to be really useful in an open environment, we are adding proper authorization mechanisms. In particular, we leverage 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 which can be mapped in JavaBeans, while they can take advantage of specialized agents, called Ontology Servers, when they need to use more complex ontologies which cannot be completely mapped in JavaBeans. 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 the improvement of our two level

software solution by trying alternative solutions to the use of the Jena software tool.

5 References

1. The Agentcities Network project home page. <http://www.agentcities.net>.
2. K Baclawski, M. K. Kokar, P. Kogut, L. Hart, J. E. Smith, J. Letkowski, and P. Emery, Extending the Unified Modeling Language for ontology development, International Journal Software and Systems Modeling (SoSyM) 1(2) (2002) 142-156.
3. Bechhofer, R. Volz, and P. Lord. Cooking the semantic web with the OWL API. In Proc. Int Semantic Web Conference, pages 659 - 675, Sanibel Island, FL, 2003.
4. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson. Jena: Implementing the Semantic Web Recommendations. In Proc 13th Int World Wide Web Conference, pages 74-83, New York, NY, 2004.
5. The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, 2002.
6. 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>
7. JADE software and documentation. <http://jade.tilab.com>.
8. Janino software and documentation. <http://janino.net>.
9. Jena, HP Labs Semantic Web Toolkit software and documentation. <http://jena.sourceforge.net/>
10. A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of owl ontologies into java. In Proceedings of Software Engineering and Knowledge Engineering Conference. (SEKE) 2004, Banff, Canada, 2004.
11. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997
12. OWL API software and documentation. <http://owl.man.ac.uk/api.shtml>
13. Velocity software and documentation. Available from <http://jakarta.apache.org/velocity>. Word Wide Web Consortium (W3C). OWL . Web Ontology Language. <http://www.w3.org/TR/owl-ref>

On the Cost of Agent-awareness for Negotiation Services

Andrea Giovannucci

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

Juan A. Rodríguez-Aguilar

Artificial Intelligence Research Institute, IIIA
Spanish Council for Scientific Research, CSIC
08193 Bellaterra, Barcelona, Spain
jar@iia.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 at shedding some light on this matter.

1. Introduction

While a significant number of agent-based applications for electronic commerce has 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 intends to shed some light on this matter. We depart from *iBundler* (fully described in [5]), an agent-aware negotiation service for combinatorial negotiations designed to be employed as: (1) an open agent platform within the Agentcities.RDT¹ (<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*[12], a commercial negotiation tool, to allow for

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.

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 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 organised as follows. First, 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 focuses on scalability issues as robustness, system performance with large populations of agents and ontology engineering. Brazier et al. [2] address the problem of scalability in naming services and location services. Besides, they analyse the concept of scalability in multi agent systems (MAS) and discuss scalability for many existing multi-agent frameworks. Deters [3] studies the problems

derived from large number of agents running in a MAS, agent resource consumption, the exchange of great number of messages, identifying agent hosting and message routing as bottle-necks. Furthermore, he performs some scalability experiments. An important result in [3] is that the main deficiencies of JESS (<http://herzberg.ca.sandia.gov/jess/>) derive from serialisation processes. Kahn investigates how timing of sequential agent registration and lookup varies as the total number of registered agents increases in COABS [8]. The works in [9] and [4] analyse robustness and fault tolerance, whereas [15] exemplifies ad-hoc, domain-dependent agent technology scaling techniques. On the other hand, the literature on ontology scalability focuses on three major issues: the size of ontology contents, the complexity of ontology construction and knowledge re-usability ([7], [14]). In particular, Jarrar states that experience shows that "unscalable solutions emerging from academic research often fails at the industrial level" [7].

Thus, we believe that it is an urging 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 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, that is surely the most widely adopted by the agent community². Secondly, the considerations emerging from the experiments derived in this paper are related to 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

² OGM (www.ogm.org) is another standardisation 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].

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 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[13] respectively. To the best of our knowledge, *iBundler* represents the first agent-aware service for multi-item negotiations, since agent services have mostly focused 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* [12] 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.

Figure 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 capable of fulfilling them; collection of bids; winner determination in a combinatorial negotiation/auction; and award

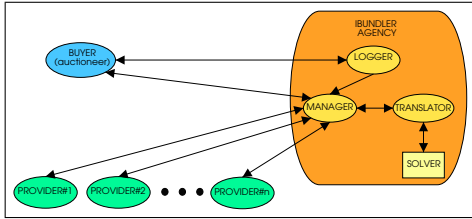


Figure 1. Architecture of the *iBundler* Agency

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 a Java MIP modeller that integrates the GNU Programming Kit GLPK (<http://www.gnu.org/directory/GNU/glpk.html>)). The Solver component is complete in the meaning 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, which one depending on the underlying branch-and-bound algorithm ([6]).

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. Besides, 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 bottle-neck in the overall process when many buyers access the service concurrently. Such limitation could be overcome by creating multiple instances of Translator Agents and Solvers on different machines. Anyway in this work we focused on the

service performances in managing big size negotiation scenarios, not on multiple concurrent accesses to the service. We leave such issue as a possible future development.

Figure 2 depicts the interaction protocol involved in the interplay of buyers and provides with *iBundler*. It is expressed in AUML (Agent Unified Modelling Language)[10] following the FIPA interaction protocol library specification compiled in [1]. Observe that the specification in figure 2 involves four roles, namely: buyer, manager, translator, and provider. Whereas multiple agents can act as providers, the remaining roles can be uniquely adopted by a single agent each. Notice too that the *iBundler* interaction protocol is composed of several interleaved interaction protocols:

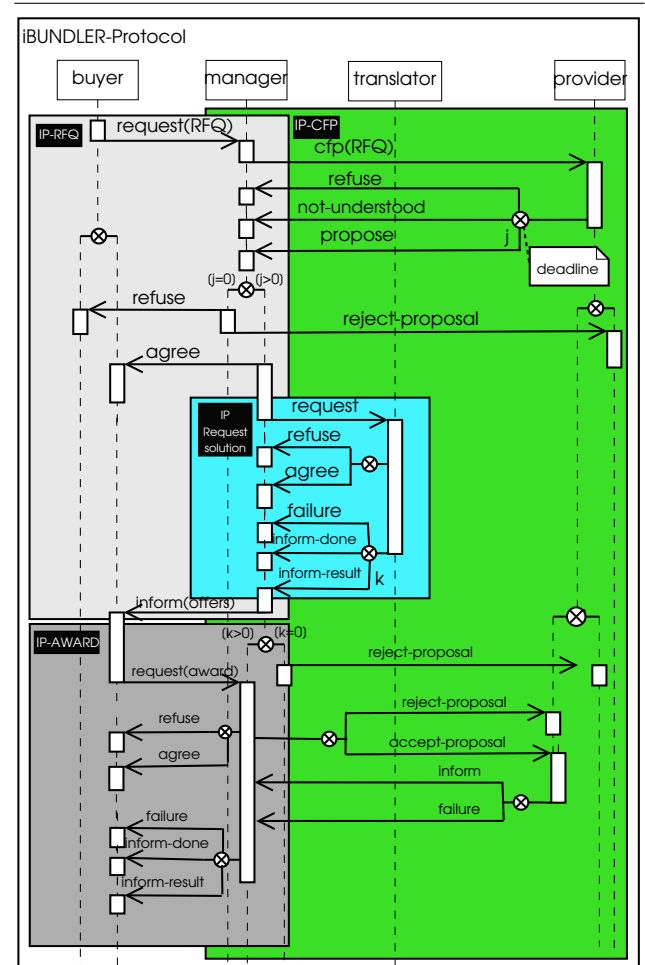


Figure 2. *iBundler* Interaction Protocol

[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 to obtain the

optimal set of offers from the available providers. In case 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 out an instance of the IP-Request-Solution interaction protocol. The protocol winds up 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.

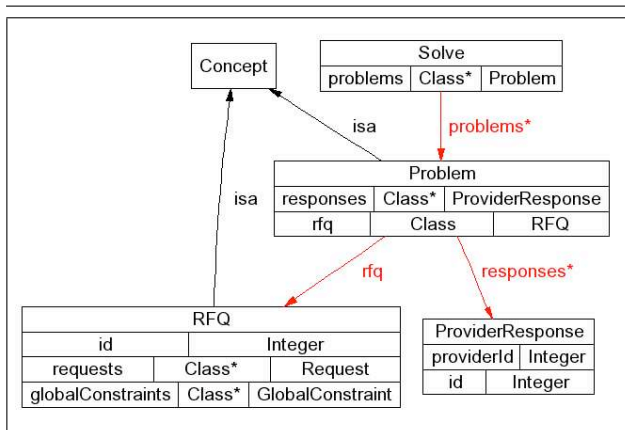


Figure 3. Problem concept

[IP-Request Solution] This interaction protocol held between the manager and the translator agent within the *iBundler* agency aims 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 may request also to receive all offers. Thereafter, if the buyer received a non-empty optimal set of offers ($k_i > 0$ in figure 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, figure 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 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 propose 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.

Once the manager agent collects all offers submitted by providers, he 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 request in IP-Request-Solution). Finally, the solution produced by the Solver component is transformed by the translator agent into a *Solution* concept, that is handed over to the manager (via 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 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 sim-

ulate such an ideal situation generating multiple datasets in separate files, each one standing for a different input negotiation problem composed of FIPA messages, each one 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 neither inter-item nor 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 to firstly understand how JADE manipulates messages and ontological objects. In particular we summarise the process of sending and receiving messages (for a complete description refer to the JADE documentation). Figure 4 graphically summarises 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 serialised 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.

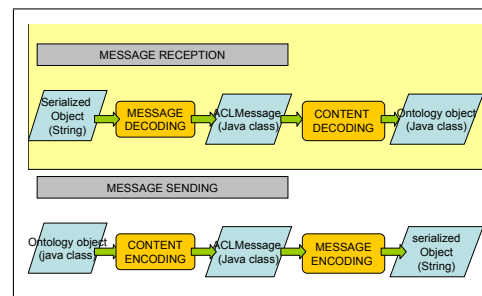


Figure 4. Message life cycle in JADE

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

Δt_1 : JADE decodes all the FIPA messages contained in the data set file containing the input negotiation problem, con-

verting them into instances of the *ACLMessage* Java class.

Δt_2 : 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.

Δt_3 : 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.

Δt_4 : 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.

Δt_5 : the *Translator* agent extracts from the received message the *Problem* ontology class containing the *RFQ* and all the collected *Bids*. Another ontology check occurs.

Δt_6 : this stage is devoted to the transformation of the *Problem* ontology class into a matrix-based format to be processed by the *Solver* component.

Δt_7 : at this stage the *Solver* component solves the MIP problem using ILOG CPLEX.

Δt_8 : the output generated by *Solver* in matrix-based format is decoded by the *Translator* agent into the *Solution* ontology class.

Δt_9 : 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.

Δt_{10} : the *Manager* agent extracts the *Solution* concept from the received *ACLMessage* with a last ontology check.

Δt_{11} : the solution is decomposed into different parts, one per provider owning an awarded bid.

Δt_{12} : 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 use 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).

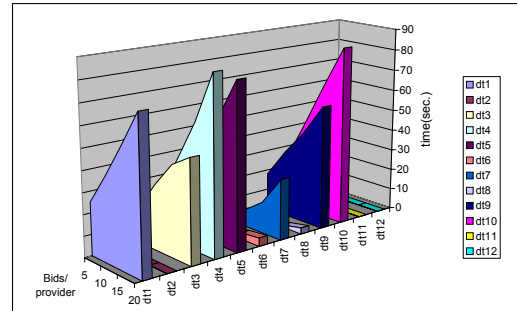


Figure 5. Time measures when varying the number of bids per provider.

Figure 5 depicts the time spent in each of the described stages, considering different number of bids per provider. We experimented similar trends varying the number of items and the number of providers³. 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 Δt_{10} is always the most time consuming, no matter the parameter being varied. Moreover, we can observe similar trends for the rest of stages (from Δt_1 to Δt_{10}). Hence, it seems that the time distribution along the different stages can be regarded as independent from the parameter setting.

Figure 6 illustrates the average percentage, over all the performed trials, of the total time that each stage consumes. We observe that: (1) The Δt_1 , Δt_3 , Δt_4 , Δt_5 , Δt_9 , Δt_{10} 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 these activities are a bottle-neck. (2) The solver time (Δt_7) is almost a negligible part of the total time. (3) Manipulating classes (stages Δt_2 , Δt_6 , Δt_8 and Δt_{11}) and solving the combina-

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

torial problem (Δt_7) is not as time-consuming as encoding and decoding messages and ontology objects.

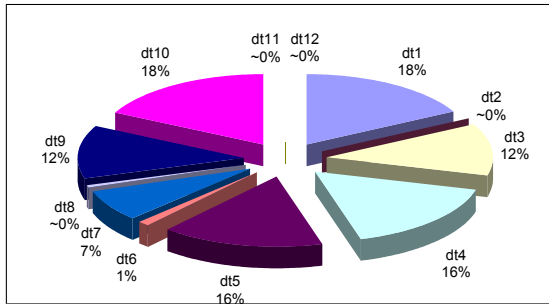


Figure 6. Average times spent at the different evaluation stages.

Figure 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, figure 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 soundly tackled with *iBundler*. Nonetheless, time performance significantly impoverishes when handling large-size negotiation scenarios.

5.2. Memory Use

In this case we found similar results when comparing the *Solver* component with *iBundler*. The amount of memory required in the worst case is quite the same for 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 represent is actually the same. The biggest amount of information is used to represent all the bids. Both *Solver* and JADE have to load in memory the information representing

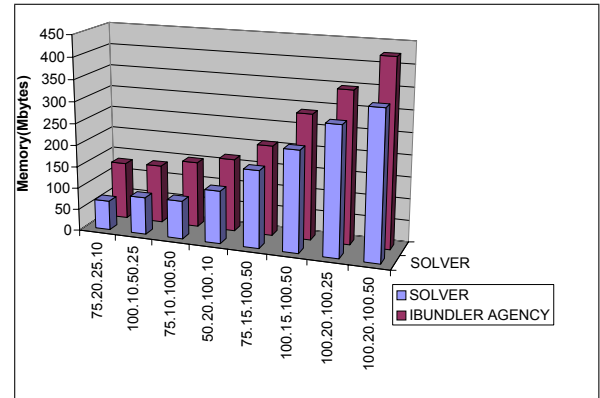


Figure 7. Memory consumption.

a problem, namely an RFQ and the received bids (the former as a Java object and the latter as a file containing matrices). Figure 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 use 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 serialisations and deserialisations, 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 serialisation 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

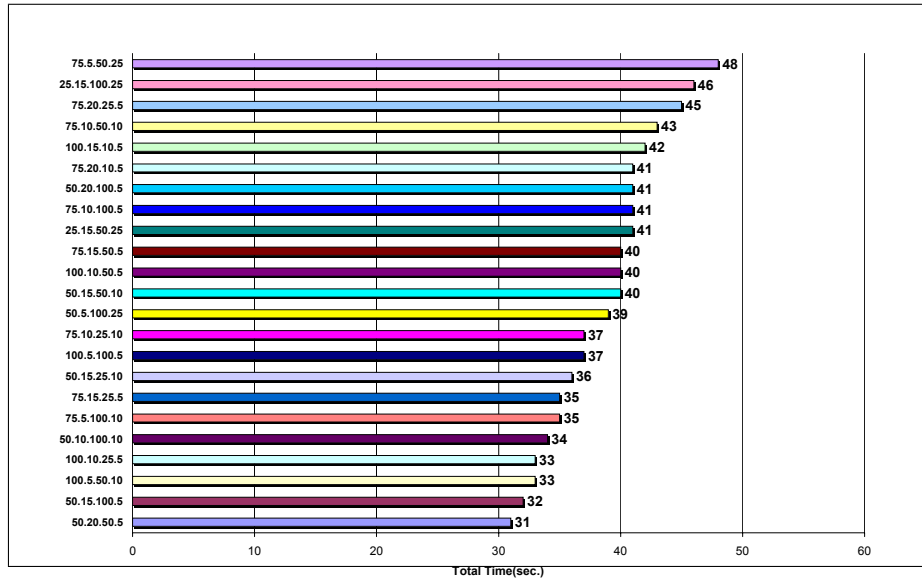


Figure 8. Time performance for negotiation scenarios on the edge of acceptability

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.

References

- [1] FIPA interaction protocol library specification. Technical Report DC00025F, Foundation for Intelligent Physical Agents.
- [2] F. Brazier, M. van Steen, and N. Wijnngaards. On MAS scalability. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pages 121–126, Montreal, May 2001.
- [3] R. Deters. Scalability & multi-agent systems. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, May 2001.
- [4] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 737–744. ACM Press, 2002.
- [5] A. Giovanucci, J. A. Rodríguez-Aguilar, A. Reyes-Moro, F. X. Noria, and J. Cerquides. Towards automated procurement via agent-aware negotiation support. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, New York, July 19-23 2004.
- [6] F. S. Hillier and G. J. Liberman. *Introduction to Operations Research*, pages 576–653. Mc Graw Hill, 2001.
- [7] M. Jarrar and R. Meersman. 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, 2002. SSGRR education center.
- [8] M. L. Kahn and C. Della Torre Cicalese. COABS grid scalability experiments. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):171–178, 2003.
- [9] M. Klein, J. Rodríguez-Aguilar, and C. Dellarocas. 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(1-2):179–189, 2003.
- [10] J. Odell, H. van Dyke Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information Systems Workshop*, pages 3–17, Austin, TX, 2000. 17th National Conference on Artificial Intelligence.
- [11] OMG and FIPA standardisation for agent technology: competition or convergence? <http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch2/ch2.htm>.
- [12] A. Reyes-Moro, J. A. Rodríguez-Aguilar, M. López-Sánchez, J. Cerquides, and D. Gutiérrez-Magallanes. Embedding decision support in e-sourcing tools: Quotes, a case study. *Group Decision and Negotiation*, 12:347–355, 2003.
- [13] T. Sandholm, S. Suri, A. Gilpin, and D. Levine. Winner determination in combinatorial auction generalizations. In *First Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 69–76, Bologna, July 2002.
- [14] H. Wache, L. Serafini, and R. García-Castro. D2.1.1 survey of scalability techniques for reasoning with ontologies. Technical report, Knowledge Web, July 2004.
- [15] M.-J. Yoo. 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*, pages 264–271. ACM Press, 2002.

Automated Interpretation of Agent Behavior

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 interpretation of agent behavior of the implemented system. This paper describes an approach and tool to automate creating interpretations of agent behavior from observations of the implementation execution, thus helping users (i.e., designers, developers, and end-users) comprehend agent behaviors. By explicitly modeling 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 behavior 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 behavior, and generating explanations.

1. Introduction

Agents are distributed software entities that are capable of autonomous decision-making. Besides being motivated by its own goals, an agent's behavior 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

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 or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

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 behavior is done manually by interpreting observations from the implementation executions. The interpretation process links (usually a causal link) observations together to create a connected, comprehensive view of what the software is doing. In essence, interpretation compares the actual implementation behavior with expected behavior, which may have been gathered 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 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 how the interpretation of agent behavior can be automated and how the Tracer Tool can be used to help build and verify the user's comprehension of the implemented agent system.

Sophisticated software such as agent systems 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 detailed abstraction level, their dependence on analyzing source code, their lack of automation to help decipher tremendous amounts of resulting data, and their lack of a model for how much the user understands. Taking the formal approach to modeling systems (and thus, understanding properties of systems), model-checking 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) modeling

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 behavior. 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 (e.g., beliefs, goals, intentions, actions, 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 behavior (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 [10]. In this research, agent concepts are used to take advantage of the user's intuitive knowledge of agent-based systems to comprehend agent behavior in the implementation.

Software comprehension, which historically has been associated with program comprehension and reverse engineering, involves extracting and representing the structural and behavioral 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) re-designed 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 [1] and PBS [3]) analyze 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" [11]. To attain an understanding of agent behavior, the models resulting from the comprehension process must be at the abstraction level where agent concepts are the elemental or base concepts.

In addition to static analysis of the source code, dynamic analysis tools (e.g., SCED [6] and Hindsight [4]) 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' runtime behavior (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 [2]) 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 runtime observations together in a causal graph. This is similar to the GUPRO toolset [7], where source code is transformed into graphs, except that the graphs nodes are in terms 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 of the actual system.

3. Building the Interpretation

When a user tries to comprehend agent behavior 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 Figure 1, background knowledge about the expected behavior 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 modeled using a semantic network (i.e., directed graph) of agent concepts that are intercon-

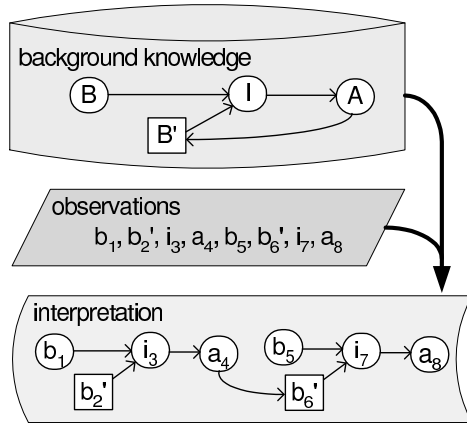


Figure 1. An interpretation for an agent, given the background knowledge K and observations O_s .

nected 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 Figure 1, the background knowledge for an agent’s behavior 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 behavior in the implemented system. Note that K represents a behavioral pattern and, thus, can have cycles in the graph. However, the interpretation, which consists of actual observations and their relationships, do 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.

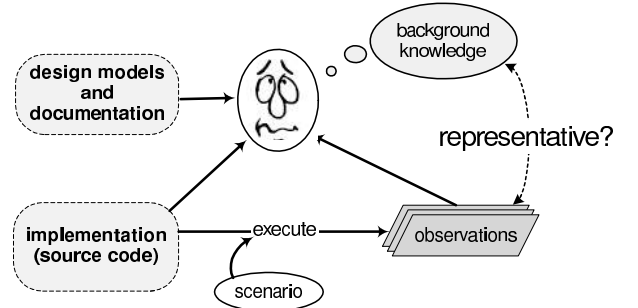


Figure 2. Manual software comprehension

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 Figure 2, the manual procedure for building comprehension can be expressed as

$$K' = \text{update}_{\text{manual}}(K, D, I, O_s) \quad (1)$$

where K is the previous background knowledge, D is the design models and documentation, I is the implementation expressed in source code, and O_s is a set of observations

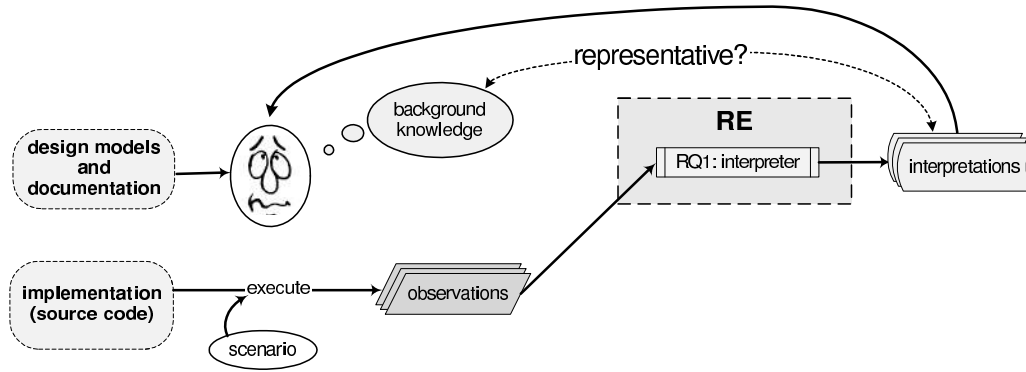


Figure 3. Reverse engineering approach

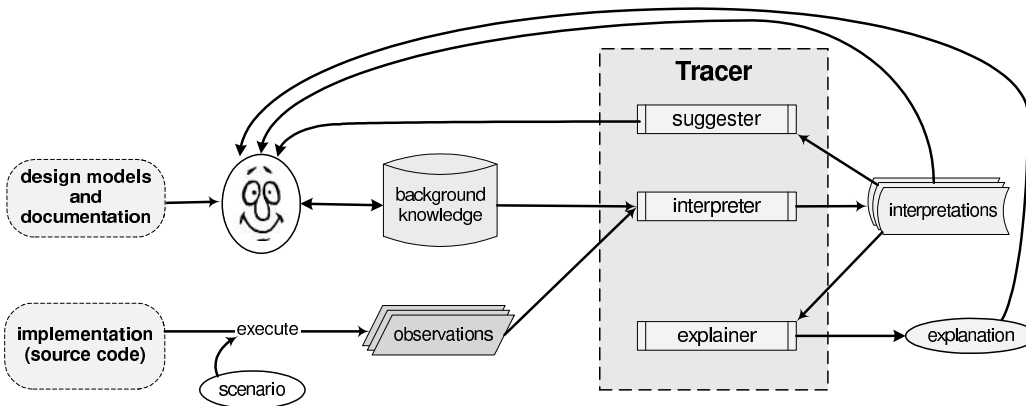


Figure 4. Automated interpretation approach using Tracer

resulting from executing the implementation I in some scenario s :

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

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 .

Due to human error or outdated design specifications, system behavior described by K may be erroneous or inaccurate with respect to the actual behavior 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 behavior. Using empirical techniques, the user must manually verify that the expected behavior expressed as K is representative of the actual behavior from the implementation. Due to complexities and uncertainties of some systems, agent behaviors cannot always be predicted from only the design

specification in general [5]. 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 behaviors 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 behavior.

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 Figure 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 behavior:

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

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 Figure 4). This is possible by explicitly modeling the user's background knowledge K and using it as input to the Tracer Tool. Thus, the new *update* function is

$$K' = update(K, D, N_s, k) \quad (4)$$

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

$$N_s = interpret(K, O_s) \quad (5)$$

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

$$k = suggest(N_s) \quad (6)$$

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 analyze the observations O_s from the implementation execution or verify that K accurately reflects the implementation's behavior, 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 behavior changes (resulting from design changes or maintenance tasks) and is different from the expected behavior represented by K , the Tracer Tool alerts the user of the new or inconsistent behavior 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 :

$$Representative(K, I, S) \iff \forall s \in S (Complete(K, N_s) \wedge Consistent(K, N_s)) \quad (7)$$

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 behaviors. 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.

4. Tracer Tool

To generate accurate interpretations, the background knowledge K should be representative of what is being explained (i.e., agent behavior in the implementation). This implies that the K (representing expected agent behavior) must be complete and consistent with the implementation's behavior (Equation 7). By explicitly modeling 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 modeled in K and ignore already modeled relations.

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

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

Since K and N_s are modeled 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

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 behaviors 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.

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

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 is 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 is analyzed, 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 Figure 4. 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 modeled 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 [9], a detailed demonstration of creating interpretations using the Tracer Tool is described.

Essentially, K is being used as a template for creating the interpretation. K is a representation of expected behavior, while N_s is a representation of actual behavior. 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 behaviors are modeled by the expected behavior representation of the background knowledge. If K is not representative of the implementation's behavior ($\neg Representative(K, I, S)$) 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 modeled in K) an agent to have a belief called ‘target location’ before creating an intention involving that target and that belief observation 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 [8].

Table 4 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 (casual) 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.

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 behavior 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 behavior (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 behavior.

Unfortunately, ensuring accurate explanations is difficult because the implementation evolves over time and there

are many factors that can influence agent behavior. First, since comprehending the behavior 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 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 Figure 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 Figure 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. If an interpretation N_s of the scenario exists, the same explanation can be generated faster using N_s because $interpret(K, O_s)$ has already done the work of mapping and relating the observations. 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 . This can be performed recursively to an arbitrary depth to find causes of causes.

$$\epsilon = explain(m, N_s) = explain(m, K, O_s) \quad (8)$$

From Equation 8, generating explanations is dependent on the quality of the K , specifically on how accurately the K reflects the context of what is being explained – thus, stressing the need to maintain the representativeness between K and I as described in Section 3.

Since background knowledge K is represented using agent concepts, the generated explanations will be in terms

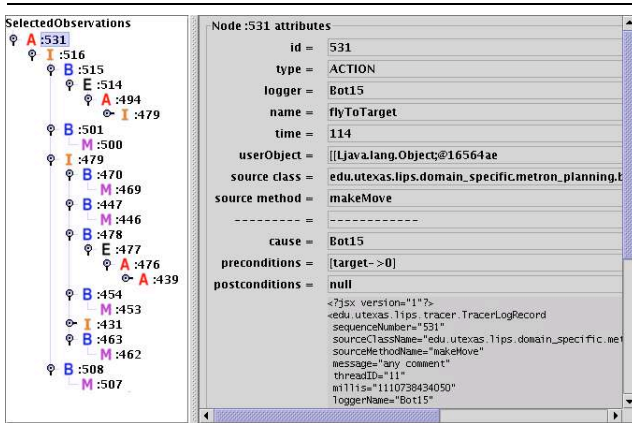


Figure 5. Explanation in Tracer

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 Figure 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 behavior. With explanations readily available to the user, tasks such as redesigning, debugging, and understanding agent behavior 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 behaviors within agent-based software systems. This paper describes an approach to automate the process of interpreting agent behavior. 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 behavior, 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 behavior, scanning for unexpected behavior, and generating explanations. With the verified background knowledge, accurate explanations of actual agent behavior 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 analyze reasons for agent behavior, thereby facilitating software maintenance tasks and promoting confidence in the adoption of agent technology.

References

- [1] A. Agrawal, M. Du, C. McCollum, T. Syst, K. Wong, P. Yu, and H. Mller. Rigi - An End-User Programmable Tool for Identifying Reusable Components. In *5th International Conference on Software Reuse*, June 2-5, 1998.
- [2] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based Speculative Parallelism. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, Monterey, California, December 10 2000. ACM Press.
- [3] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Meller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
- [4] Hindsight, 2004. <http://www.testersedge.com/hindsight.htm>.
- [5] N. R. Jennings. On Agent-based Software Engineering. *Artificial Intelligence*, 117:277–296, 2000.
- [6] K. Koskimies, T. Mnnist, T. Syst, and J. Tuomi. Automated Support for Modeling OO Software. *IEEE Software*, 15(1):87–94, 1998.
- [7] B. Kullbach and A. Winter. Querying as an Enabling Technology in Software Reengineering. In P. Nesi and C. Verhoef, editors, *3rd European Conference on Software Maintenance and Reengineering*, pages 42–50, Los Alamitos, 1999. IEEE Computer Society.
- [8] D. N. Lam and K. S. Barber. Comprehending Agent Software. In *4th International Joint Conference on Autonomous Agents and Multi-Agent Systems*, Utrecht, Netherlands, July 25-29, 2005.
- [9] D. N. Lam and K. S. Barber. Debugging Agent Behavior in an Implemented Agent System. In Bordini, Dastani, Dix, and Seghrouchni, editors, *Lecture Notes in Computer Science*, volume 3346, pages 103–125. Springer-Verlag, 2005.
- [10] E. Stroulia and T. Syst. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review*, 10(1):8–17, 2002.
- [11] M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley and Sons, Chichester, England, 2002.

Requirements Analysis of an Agent's Reasoning Capability

Tibor Bosse¹, Catholijn M. Jonker², and Jan Treur¹

¹ Vrije Universiteit Amsterdam,
Department of Artificial Intelligence
De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands
{tbosse, treur}@cs.vu.nl

² Radboud Universiteit Nijmegen, Nijmegen
Institute for Cognition and Information,
Montessorilaan 3,
6525 HR Nijmegen, The Netherlands
C.Jonker@nici.ru.nl

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 characterise 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 realised in a design and implementation of this system; e.g., [9], [16], [21]. Recently, requirements analysis for concurrent systems and agent systems has been addressed in particular, for example, in [11], [13]. 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.

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., [22]. Properties of such a functional relation can be related to properties of a knowledge base used to realise the functionality, which provides possibilities for verification and validation of this knowledge; e.g., [18]. However, more sophisticated reasoning capabilities can better be considered as involving a process over time; especially for nontrivial reasoning patterns the temporal aspects play an important role in their semantics; cf. [12], [19]. Therefore, within an agent-oriented software engineering approach to an agent's reasoning capability, requirements specification has to address 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 formalisation of reasoning processes by traces consisting of sequences of reasoning states over time, following the semantic formalisation introduced in [12]. Reasoning processes as performed by humans may involve dynamically introduced or retracted assumptions: a pattern used as a case study in this paper, further on 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 characterise the reasoning pattern.

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. Logical relationships

have been determined between dynamic properties at one aggregation level and those of a lower aggregation level. These characterising properties have been formalized using the temporal trace language TTL, thus enabling automated support of analysis. As an additional validation of this characterisation, 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 [5]. Supported by software tools, the properties were checked against the formalised scenarios of these human traces, and confirmed.

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 [2], and a dedicated software environment for simulation that has been developed [3], these executable properties were used to generate simulation traces. Moreover, for these traces the (higher-level) dynamic properties were checked and confirmed, which validates the identified logical relationships between the dynamic properties at different aggregation levels.

Finally, a design of an existing software agent performing reasoning by assumption [15] was analysed. This agent was designed using the component-based design method DESIRE [6]. 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 checked, and found confirmed.

In Section 2 the dynamic perspective on reasoning is discussed in some more detail, and 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 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 semantical formalisation of the dynamics of reasoning exploited here is based on the concepts reasoning state, transitions and traces.

Reasoning state. A reasoning state formalises 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 $\langle S, S' \rangle$ of $RS \times RS$. A *reasoning transition relation* is a set of these transitions, or a relation on $RS \times RS$ 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 time frame (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 formalises 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 characterised by requirements.

3. Dynamic Properties

To specify properties on the dynamics of reasoning, the temporal trace language TTL used in [13] is adopted. This is a language in the family of languages to which also situation calculus [20], event calculus [17], and fluent calculus [14] 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* \times *SIGN* and the relation *prediction_for* on *INFO_ELEMENT* \times *SIGN* \times *INFO_ELEMENT* \times *SIGN*. Table 1 contains all 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*, *sparkling_plugs_problem*. The sort *SIGN* consists of the elements *pos* and *neg*.

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 \models between states and state properties is used: $S \models p$ means that state property *p*

holds in state *S*. For example, in the reasoning state *S* above it holds $S \models \text{assumed}(\text{battery_empty}, \text{pos})$.

Reasoning trace. To describe dynamics, explicit reference is made to time in a formal manner. A fixed time frame *T* is assumed which 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 time frame *T* is a mapping $\gamma : T \rightarrow \text{STATES}(\text{Ont})$, i.e., a sequence of reasoning states $\gamma_t (t \in T)$ in *STATES*(*Ont*). The set of all traces over ontology *Ont* is denoted by $\Gamma(\text{Ont})$, i.e., $\Gamma(\text{Ont}) = \text{STATES}(\text{Ont})^T$. The set $\Gamma(\text{Ont})$ is also denoted by Γ if no confusion is expected. 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 \models 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 $\text{state}(\gamma, t) \models p$, 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 \models 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.

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

Table 1 State ontology for the pattern ‘reasoning by assumption’

Simulation. A simpler temporal language has been used to specify simulation models. This temporal language, the LEADSTO language [3], offers the possibility to model 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 $\alpha \rightarrow_{e, f, g, h} \beta$, 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 [3]. 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 Characterising 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 Initialisation)

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. Formalisation:

$\text{initial_assumption}(A, S) \rightarrow_{0,0,1,1} \text{assumed}(A, S)$

LP2 (Prediction Effectiveness)

Local property LP2 expresses that for each assumption that is made, all relevant predictions are generated.

Formalisation:

$\text{assumed}(A, S1) \text{ and } \text{domain_implies}(A, S1, P, S2) \rightarrow_{0,0,1,1} \text{prediction_for}(P, S2, A, S1)$

LP3 (Observation Initiation Effectiveness)

Local property LP3 expresses that all predictions made will be observed. Formalisation:

$\text{prediction_for}(P, S1, A, S2) \rightarrow_{0,0,1,1} \text{to_be_observed}(P)$

LP4 (Observation Result Effectiveness)

Local property LP4 expresses that, if an observation is made the appropriate observation result will be received.

Formalisation:

$\text{to_be_observed}(P) \text{ and } \text{holds_in_world}(P, S) \rightarrow_{0,0,1,1} \text{observation_result}(P, S)$

LP5 (Evaluation Effectiveness)

Local property LP5 expresses that, if an assumption was made and a related prediction is falsified by an observation result, then the assumption is rejected.

Formalisation:

$\text{assumed}(A, S1) \text{ and } \text{prediction_for}(P, S2, A, S1) \text{ and } \text{observation_result}(P, S3) \text{ and } S2 \neq S3 \rightarrow_{0,0,1,1} \text{rejected}(A, S1)$

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. Formalisation:

$\text{assumed}(A, S1) \text{ and } \text{rejected}(A, S1) \text{ and } \text{alternative_for}(B, S2, A, S1) \text{ and } \text{not_rejected}(B, S2) \rightarrow_{0,0,1,1} \text{assumed}(B, S2)$

LP7 (Assumption Persistence)

Local property LP7 expresses that assumptions persist as long as they are not rejected. Formalisation:

$\text{assumed}(A, S) \text{ and } \text{not_rejected}(A, S) \rightarrow_{0,0,1,1} \text{assumed}(A, S)$

LP8 (Rejection Persistence)

Local property LP8 expresses that rejections persist.

Formalisation:

$\text{rejected}(A, S) \rightarrow_{0,0,1,1} \text{rejected}(A, S)$

LP9 (Observation Result Persistence)

Local property LP9 expresses that observation results persist. Formalisation:

$\text{observation_result}(P, S) \rightarrow_{0,0,1,1} \text{observation_result}(P, S)$

Using the software environment that is described in [3], 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 Figure 1. Here, time is on the horizontal axis, and the state properties and 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 Figure 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 burning), this assumption is rejected. Instead, a second assumption is made (there is a sparking plugs problem), which turns out to be correct.

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):

GP1 (Reasoning Termination)

Eventually there is a time point at which the reasoning terminates.

$$\forall \gamma : \Gamma \exists t : T \text{ termination}(\gamma, t)$$

Here $\text{termination}(\gamma, t)$ is defined as follows:

$$\forall t' : T \quad t' \geq t \Rightarrow \text{state}(\gamma, t) = \text{state}(\gamma, t')$$

GP2 (Correctness of Rejection)

Everything that has been rejected does not hold in the world situation.

$$\forall \gamma : \Gamma \forall t : T \forall A : \text{INFO_ELEMENT} \forall S : \text{SIGN} \\ \text{state}(\gamma, t) \models \text{rejected}(A, S) \Rightarrow \\ \text{state}(\gamma, t) \not\models \text{holds_in_world}(A, S)$$

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.

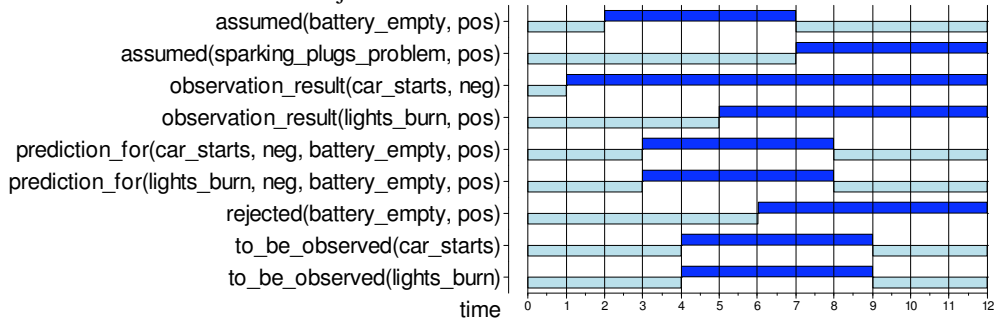


Figure 1 Example simulation trace

$$\forall \gamma : \Gamma \forall t : T \text{ termination}(\gamma, t) \\ \Rightarrow [\exists A : \text{INFO_ELEMENT}, \exists S : \text{SIGN} \\ \text{state}(\gamma, t) \models \text{assumed}(A, S) \wedge \text{state}(\gamma, t) \not\models \text{rejected}(A, S)]$$

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

WP1 (Static World)

If something holds in the world, it will hold forever.

$$\forall \gamma : \Gamma \forall t : T \forall A : \text{INFO_ELEMENT} \forall S : \text{SIGN} \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S) \Rightarrow \\ [\forall t' : T \geq t : T \text{ state}(\gamma, t') \models \text{holds_in_world}(A, S)] \\ \forall \gamma : \Gamma \forall t : T \forall A : \text{INFO_ELEMENT} \forall S : \text{SIGN} \\ \text{state}(\gamma, t) \not\models \text{holds_in_world}(A, S) \Rightarrow \\ [\forall t' : T \geq t : T \text{ state}(\gamma, t') \not\models \text{holds_in_world}(A, S)]$$

WP2 (World Consistency)

If something holds in the world, then its complement does not hold.

$$\forall \gamma : \Gamma \forall t : T \forall A : \text{INFO_ELEMENT} \forall S1, S2 : \text{SIGN} \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S1) \wedge S1 \neq S2 \Rightarrow \\ \text{state}(\gamma, t) \not\models \text{holds_in_world}(A, S2)$$

DK1 (Domain Knowledge Correctness)

The domain-specific knowledge is correct in the world.

$$\forall \gamma : \Gamma \forall t : T \forall A, B : \text{INFO_ELEMENT} \forall S1, S2 : \text{SIGN} \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S1) \wedge \text{domain_implies}(A, S1, B, S2) \\ \Rightarrow \text{state}(\gamma, t) \models \text{holds_in_world}(B, S2)]$$

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.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A: \text{INFO_ELEMENT} \forall S1: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{rejected}(A, S1) \Rightarrow \\ [\exists t': T \leq t: T \exists B: \text{INFO_ELEMENT} \exists S2, S3: \text{SIGN} \\ \text{state}(\gamma, t') \models \text{prediction_for}(B, S2, A, S1) \wedge \\ \text{state}(\gamma, t') \models \text{observation_result}(B, S3) \wedge S2 \neq S3] \end{aligned}$$

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.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A, B: \text{INFO_ELEMENT} \forall S1, S2, S3: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{prediction_for}(B, S2, A, S1) \wedge \\ \text{state}(\gamma, t) \models \text{observation_result}(B, S3) \wedge S2 \neq S3 \Rightarrow \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S1) \end{aligned}$$

IP3 (Observation Result Correctness)

Observation results obtained from the world indeed hold in the world.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A: \text{INFO_ELEMENT} \forall S: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{observation_result}(A, S) \Rightarrow \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S) \end{aligned}$$

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.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A, B: \text{INFO_ELEMENT} \forall S1, S2, S3: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{prediction_for}(B, S2, A, S1) \wedge \\ \text{state}(\gamma, t) \models \text{holds_in_world}(B, S3) \wedge S2 \neq S3 \Rightarrow \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S1) \end{aligned}$$

IP5 (Observation Result Grounding)

If an observation has been obtained, then earlier on the corresponding fact held in the world.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A: \text{INFO_ELEMENT} \forall S: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{observation_result}(A, S) \Rightarrow \\ [\exists t': T \leq t: T \text{state}(\gamma, t') \models \text{holds_in_world}(A, S)] \end{aligned}$$

IP6 (Incorrect Prediction implies Incorrect Assumption 2)

If a prediction does not hold in the world, then the associated assumption does not hold either.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A, B: \text{INFO_ELEMENT} \forall S1, S2: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{prediction_for}(B, S2, A, S1) \wedge \\ \text{state}(\gamma, t) \not\models \text{holds_in_world}(B, S2) \Rightarrow \\ \text{state}(\gamma, t) \not\models \text{holds_in_world}(A, S1) \end{aligned}$$

IP7 (Prediction Correctness)

If a prediction is made for an assumption that holds in the world, then the prediction also holds.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A, B: \text{INFO_ELEMENT} \forall S1, S2: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{prediction_for}(B, S2, A, S1) \wedge \\ \text{state}(\gamma, t) \models \text{holds_in_world}(A, S1) \Rightarrow \\ \text{state}(\gamma, t) \models \text{holds_in_world}(B, S2) \end{aligned}$$

5. Relationships Between Dynamic Properties

A number of logical relationships have been identified between properties at different aggregation levels. An overview of all identified logical relationships relevant for GP2 is depicted as an AND-tree in Figure 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.

$$\begin{aligned} \forall \gamma, \Gamma \forall t: T \forall A, B: \text{INFO_ELEMENT} \forall S1, S2: \text{SIGN} \\ \text{state}(\gamma, t) \models \text{prediction_for}(B, S2, A, S1) \Rightarrow \\ [\exists t': T \leq t: T \text{state}(\gamma, t') \models \text{assumed}(A, S1) \wedge \\ \text{domain_implies}(A, S1, B, S2)] \end{aligned}$$

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.

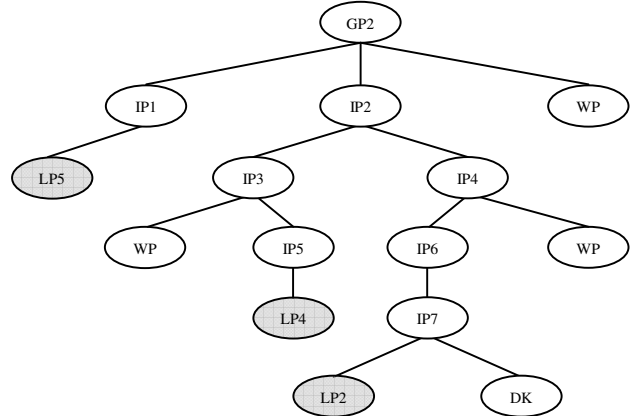


Figure 2 AND-Tree of Dynamic Properties

The relationships depicted in Figure 2 should be interpreted as semantic entailment relationships. For example, the relationship at the highest level expresses that the implication $IP1 \ \& \ IP2 \ \& \ WP1 \Rightarrow GP2$ 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 [5].

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.

	Human Traces (Taken from [5])	Simulation Traces (This paper)	Prototype Traces (Taken from [15])
Local Properties	+/-	+	+
Intermediate Properties	+/-	+	+
Global Properties	+/-	+	+

Table 2 Overview of the different verification results

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 [5].

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 formalisation of the properties. Checking the properties against simulation traces turned out to be useful to localise 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, see [15]. This agent was designed on the basis of the component-based design method DESIRE, cf. [6]. 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 enabled the results of the requirements elicitation and specification phase to be formally verified and improved.

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 [19]. 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 language TTL used here allows for precise specification of these dynamic properties, 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 traces (in the context of prototyping) and empirical traces (in the context of requirements elicitation and validation in cooperation 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 to requirements analysis of 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 [4]. 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.* [6]. However, GDM is just one possible approach for Agent-Oriented Software Engineering. Recently, several other architectures have been proposed, for example, Tropos [8], KAOS [10] or GBRAM [1]. In future work, the possibilities may be explored to incorporate the approach based on dynamic properties presented here within such architectures. Especially for architectures that provide a specific language for formalisation of requirements (KAOS for example uses a real-time temporal logic to specify requirements in terms of goals, constraints and objects), these possibilities are promising.

References

- [1] Antón, A.I. (1996). Goal-based Requirements Analysis, *Proc. of the International Conference on Requirements Engineering (ICRE'96)*, IEEE Computer Soc. Press, Colorado Springs, Colorado, USA, pp. 136- 144.
- [2] Barringer, H., Fisher, M., Gabbay, D., Owens, R., and Reynolds, M. (1996). *The Imperative Future: Principles of Executable Temporal Logic*, Research Studies Press Ltd. and John Wiley & Sons.
- [3] Bosse, T., Jonker, C. M., van der Meij, L., and Treur, J. (2005). LEADSTO: a Language and Environment for Analysis of Dynamics by SimulaTiOn (extended abstract). *Proc. of the 18th International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems, IEA/AIE 2005*. LNAI, Springer Verlag. In press.
- [4] Bosse, T., Jonker, C.M., and Treur, J. (2004). 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, pp. 293-297.
- [5] Bosse, T., Jonker, C.M., and Treur, J. (2005). Reasoning by Assumption: Formalisation and Analysis of Human Reasoning Traces. In: *Proceedings of the First International Work-conference on the Interplay between Natural and Artificial Computation, IWINAC'05*. LNAI, vol. 3561, Springer Verlag, pp. 430-439.
- [6] Brazier, F.M.T., Jonker, C.M., and Treur, J. (2002). Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, vol. 41, pp. 1-28.
- [7] Brazier F.M.T., Langen P.H.G. van, Treur J. (1996). A logical theory of design. In: J.S. Gero (ed.), *Advances in Formal Design Methods for CAD, Proc. of the Second International Workshop on Formal Methods in Design*. Chapman & Hall, New York, pp. 243-266.
- [8] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2004). Tropos: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agent and Multi-Agent Systems*, vol. 8, pp. 203-236.
- [9] Dardenne, A., Lamsweerde, A. van, and Fickas, S. (1993). Goal-directed Requirements Acquisition. *Science in Computer Programming*, vol. 20, pp. 3-50.
- [10] Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1998). GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering, *Proc. ICSE'98 - 20th International Conference on Software Engineering*, Kyoto, vol. 2, pp. 58-62.
- [11] Dubois, E., Du Bois, P., and Zeippen, J.M. (1995). A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems. In: *Proceedings of the Real-Time Systems Conference, RTS'95*.
- [12] Engelfriet, J., and Treur, J. (1995). Temporal Theories of Reasoning. *Journal of Applied Non-Classical Logics*, 5, pp. 239-261.
- [13] Herlea, D.E., Jonker, C.M., Treur, J., and Wijngaards, N.J.E. (1999). 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, pp. 8-27.
- [14] Hölldobler, S., and Thielscher, M. (1990). A new deductive approach to planning. *New Generation Computing*, 8:225-244.
- [15] Jonker, C.M., and Treur, J. (2003). Modelling the Dynamics of Reasoning Processes: Reasoning by Assumption. *Cognitive Systems Research Journal*, vol. 4, pp. 119-136.
- [16] Kontonya, G., and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley and Sons, New York.
- [17] Kowalski, R., and Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, 4, pp. 67-95.
- [18] Leemans, N.E.M., Treur, J., and Willems, M. (2002). A Semantical Perspective on Verification of Knowledge. *Data and Knowledge Engineering*, vol. 40, pp. 33-70.
- [19] Meyer, J.-J. Ch., and Treur, J. (eds.) (2001). *Dynamics and Management of Reasoning Processes*. Series in Defeasible Reasoning and Uncertainty Management Systems (D. Gabbay, Ph. Smets, series eds.), Kluwer Acad. Publishers.
- [20] Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.
- [21] Sommerville, I., and Sawyer P. (1997). *Requirements Engineering: a good practice guide*. John Wiley & Sons, Chichester, England.
- [22] Treur, J. (2002). Semantic Formalisation of Interactive Reasoning Functionality. *International Journal of Intelligent Systems*, vol. 17, pp. 645-686.

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

B. Henderson-Sellers,
J. Debenham, N. Tran
University of Technology, Sydney
brian.debenham@it.uts.edu.au;
numitran@yahoo.com

M. Cossentino
ICAR - Consiglio
Nazionale Ricerche
cossentino@pa.icar.
cnr.it

G. Low
University of New South Wales
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.

Keywords: method engineering, agent-oriented methodology, reuse, case 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-under-

pinned 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.

The analysis of PASSI discussed here is the next 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. We then ask for each fragment 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 Process for Agent Societies Specification and Implementation) [8-10] offers a step-by-step requirement-to-code process for the development of an MAS (Figure 1), integrating models and concepts from both the object-oriented (OO) software engineering and the agent-oriented paradigms. The methodology adopts (and largely extends/adapts) the UML notation for its work products and targets the FIPA implementation environment.

2.2 OPF

OPEN (Object-oriented Process, Environment and Notation) [11] is an established approach for developing software, primarily that with an object-oriented implementation. Within the OPEN approach, the most relevant element is the OPF, which comprises a metamodel that defines all the methodology¹ elements at a high level of abstraction plus a repository that contains instances of those metalevels concepts supplemented by a set of construction guidelines (Figure 2).

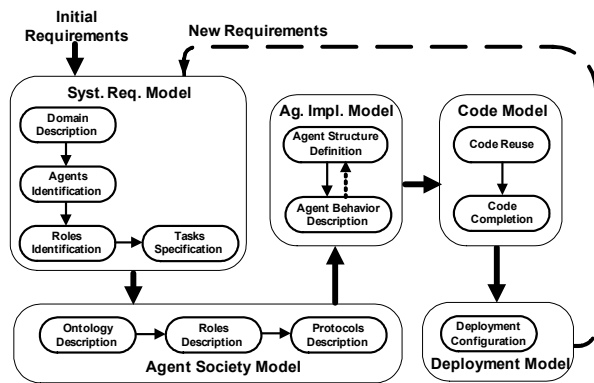


Figure 1 – Overview of PASSI

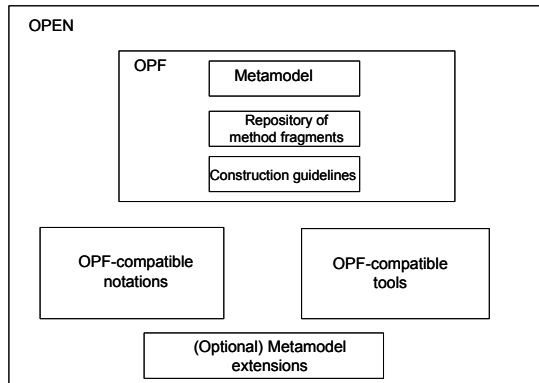


Figure 2 The OPF consists of a metamodel, a repository and construction guidelines. OPEN consists of the OPF, OPF-compatible notations and tools and optionally metamodel extensions.

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

Individual OPEN-compliant processes can then be constructed, using the method engineering approach, from the appropriate method fragments of the repository – see the example constructed process in [7].

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

3. Method Fragments in PASSI

In this section, we analyze 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*".

3.2.2. 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 Analyze requirements Use case modelling
Agent identification	Construct the agent model [12-14]
Role identification	Model agent's roles [15]
Task specification	Construct the agent model Model agents' tasks (new here)
Ontology description	Define ontologies [16] Construct the agent model
Role description	Model agents' roles [15]
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 and 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 analyze 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*” [15].

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*” [12] 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*” [15] 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*” [13] 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*” [12]. 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 (Figure 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 analyzed [17], 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 [12-16].

3.4 Fragments for Work Products

All work products of PASSI are represented in UML notation although with some 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.
- *Agent Identification Diagram:* One or more use cases in the above use case diagrams are grouped into stereotyped packages to form Agent Identification Diagrams (Figure 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.

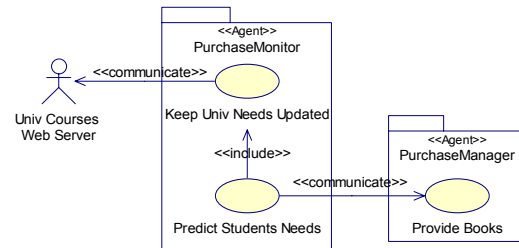


Figure 3 Agent Identification Diagram

- *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 (Figure 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.

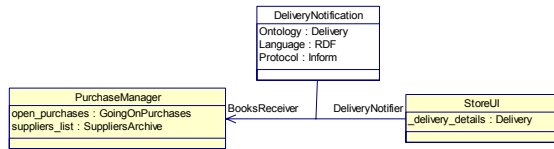


Figure 4 Communication Ontology Description Diagram

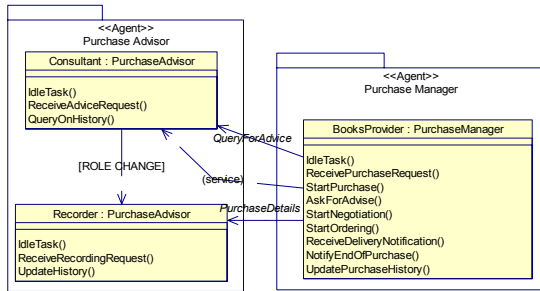


Figure 5 Role Description Diagram

- **Role Description Diagram:** This is a UML class diagram which shows agents as packages, and agents' roles as classes (Figure 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.
- **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:

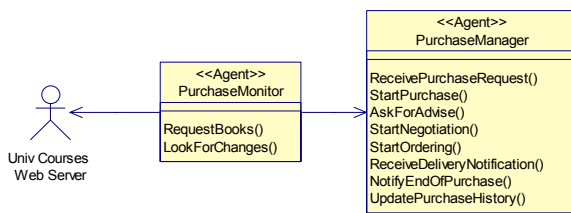


Figure 6 Multi-Agent Structure Definition Diagram

- **Multi-Agent Structure Definition Diagram:** This is a UML class diagram where classes represent agents and associations between classes signify inter-agent communications (Figure 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 focuses 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. [18]), 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.

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.

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

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 [12-17], 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 [19] 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 [20]. It is also interesting to note that the work reported here represents an evaluation of the possibilities of interaction of the FIPA Methodology TC² 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.

5. References

[1] Kumar, K. and Welke, R.J., 1992, 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, 257-269.
 [2] Brinkkemper, S., 1996, Method engineering: engineering of information systems development methods and tools, *Inf. Software Technol.*, **38(4)**, 275-280.
 [3] Ralyté, J. and Rolland, C., 2001, 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, 267-283.
 [4] Rolland, C., Prakash, N. and Benjamin, A., 1999, A multi-model view of process modelling, *Req. Eng. J.*, **4(4)**, 169-187
 [5] van Slooten, K. and Hodes, B., 1996, 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, 29-44.
 [6] Ralyté, J., 2004, 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, 271-282.
 [7] Henderson-Sellers, B., Serour, M., McBride, T., Gonzalez-Perez, C. and Dagher, L., 2004, Process construction and customization, *J. Universal Computer Science*, **10(4)**, 326-358
 [8] Burrato, P. and Cossentino, M., 2002, 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)*, May 2002, Toronto
 [9] Cossentino, M., 2005, From requirements to code with the PASSI methodology, in *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P. Giorgini), Idea Group, 79-106.
 [10] PASSI website. <http://mozart.csai.unipa.it/passi/>
 [11] Firesmith, D.G. and Henderson-Sellers, B., 2002, *The OPEN Process Framework*, Addison-Wesley, UK.
 [12] Tran, Q.N., Henderson-Sellers, B. and Debenham, J. 2004. Incorporating the elements of the MASE methodology into Agent OPEN. *Procs. 6th Int. Conference on Enterprise Information Systems (ICEIS'2004)*, 380-388.
 [13] Henderson-Sellers, B., Debenham, J., and Tran, Q.N. 2004. 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, 98-111.
 [14] Henderson-Sellers, B., Tran, Q.N.N. and Debenham, J. 2004. 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, 370-385
 [15] Henderson-Sellers, B. and Debenham, J., 2003. Towards OPEN methodological support for agent-oriented systems development. *Procs. 1st International Conference on Agent-Based Technologies and Systems*, 14-24.
 [16] Henderson-Sellers, B., Tran, Q.N.N., Debenham, J. and Gonzalez-Perez, C., 2005. 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, 149-160.
 [17] Henderson-Sellers, B., 2005, 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, 368-397.
 [18] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A., 2004, Tropos: an agent-oriented software development methodology, *Autonomous Agents and Multi-Agent Systems*, **8(3)**, 203-236.
 [19] Henderson-Sellers, B., Simons, A.J.H. and Younessi, H., 1998, *The OPEN Toolbox of Techniques*, Addison-Wesley, UK
 [20] Zhang, T.I., Kendall, E. and Jiang, H., 2002, 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, 32-46

² <http://www.pa.icar.cnr.it/~cossentino/FIPAmeth/>