W3 - The 1st International Workshop on Engineering Multi-Agent Systems (EMAS 2013)

Massimo Cossentino (National Research Council, Italy)
Amal El Fallah Seghrouchni (LIP6 - Univ. Pierre and Marie Curie, France)
Michael Winikoff (University of Otago, New Zealand)

May 6 and May 7
Scope and Theme

Although much progress has been made, the design, implementation and deployment of multi-agent systems still poses many challenges. Some of these concern design and software engineering aspects, for example, how to effectively design agents and their interactions? Other challenges concern implementation, for instance, how to effectively implement multi-agent coordination or organisations? Further challenges concern use of logic-based techniques for verification of agent systems.

It is increasingly apparent that there are benefits in considering design and implementation challenges together. For example, design artefacts can be used to support and assist with debugging and testing. Another example is the development of agent-oriented programming languages that result in programs that are more readily verifiable. A final example is the use of declarative techniques that span design and implementation. This unveils a tight interlacement among the different research issues in multi-agent systems engineering.

This naturally results in a workshop that brings together the currently separate topics (but overlapping communities) that focus on software engineering aspects (AOSE), programming aspects (ProMAS), and the application of declarative techniques to design, programming and verification (DALT).

Furthermore, a natural complement to research papers on engineering multi-agent systems is application papers that describe developed applications, and articulate lessons learned and engineering challenges that were identified in building and deploying the applications.

The EMAS workshop thus explicitly pursues three goals:

1. To progress and further develop our understanding of how to engineer multi-agent systems.
2. To bring together the communities that are concerned with different aspects of engineering multi-agent systems, and by doing so, allow for better interchange of ideas between the communities, thus exploiting the synergies discussed above.
3. To provide a venue where people who have developed applications can articulate the lessons learned and engineering challenges identified in building and deploying their applications, and have these lessons influence the ongoing research in the field.

The call for papers explicitly addressed both application and research papers that are concerned with any aspect of the engineering of multi-agent systems, specifically including any topics that would fall within the scope of one or more of the three parent workshops:

- Agent-Oriented Software Engineering (AOSE),
- Declarative Agent Languages and Technologies (DALT),
- Programming Multi-Agent Systems (ProMAS).
EMAS 2013 received 31 submissions (one was withdrawn before being reviewed). Each paper was reviewed by three reviewers, and we accepted 19 papers, including two application papers. Note that these accepted papers are a mixture of well-developed mature work, and somewhat preliminary work. Due to timing constraints the papers in these informal proceedings were not significantly revised from the submitted versions. These papers will be subsequently revised (and re-reviewed) for the Springer LNAI post-proceedings.

The EMAS 2013 chairs would like to acknowledge the great review work done by members of the Program Committee. Reviews were in general detailed (and, we hope, useful to the authors), and there was a very high degree of consensus amongst the reviewers.

We now look forward to participating in an exciting event where interesting discussions will take place about the different aspects of engineering multi-agent systems.

March 18, 2013

Massimo Cossentino
Amal El Fallah Seghrouchni
Michael Winikoff
(EMAS 2013 Workshop Chairs)
Committees

Workshop Chairs

Massimo Cossentino (National Research Council, Italy)
Amal El Fallah Seghrouchni (LIP6 - Univ. Pierre and Marie Curie, France)
Michael Winikoff (University of Otago, New Zealand)

Steering Committee

EMAS is overseen by a (merged) steering committee from the three “parent” workshops (“*” indicates chairs for the 2013 workshop, original steering committee indicated in brackets).
Matteo Baldoni (DALT; Italy)
Rafael Bordini (ProMAS; Brazil)
Mehdi Dastani (ProMAS; Netherlands)
Jürgen Dix (ProMAS; Germany)
Amal El Fallah Seghrouchni* (ProMAS; France)
Paolo Giorgini (AOSE; Italy)
Jörg Müller (AOSE; Germany)
M. Birna Van Riemsdijk (DALT; Netherlands)
Tran Cao Son (DALT; USA)
Gerhard Weiss (AOSE; Netherlands)
Danny Weyns (AOSE; Sweden)
Michael Winikoff* (DALT & AOSE; New Zealand).

Scientific Program Committee Members

Natasha Alechina (Nottingham, UK)
Matteo Baldoni (Torino, Italy)
Cristina Baroglio (Torino, Italy)
Jeremy Baxter (QinetiQ, UK)
Olivier Boissier (Saint-Etienne, France)
Rafael Bordini (FACIN-PUCRS, Brazil)
Lars Braubach (Hamburg, Germany)
Rem Collier (Dublin, Ireland)
Mehdi Dastani (Utrecht, Netherlands)
Scott DeLoach (Kansas state, USA)
Louise Dennis (Liverpool, UK)
Virginia Dignum (Delft, Netherlands)
Jürgen Dix (Clausthal, Germany)
Giancarlo Fortino (Uni. Calabria, Italy)
Aditya Ghose (Wollongong, Australia)
Paolo Giorgini (Trento, Italy)
Adriana Giret (Valencia, Spain)
Marie-Pierre Gleizes (IRIT, Uni. Paul Sabatier, France)
Jorge J. Gomez-Sanz (Madrid, Spain)
Christian Guttmann (IBM, Australia)
James Harland (RMIT, Australia)
Vincent Hilaire (Belford-Montbelliard, France)
Koen Hindriks (Delft, Netherlands)
Benjamin Hirsch (Berlin, Germany)
Tom Holvoet (KU Leuven, Belgium)
Jomi Hübner (Federal University of Santa Catarina, Brazil)
Michael Huhns (South Carolina, USA)
Joao Leite (Lisboa, Portugal)
Yves Lesperance (York, Canada)
Brian Logan (University of Nottingham, UK)
Viviana Mascardi (Genova, Italy)
Philippe Mathieu (Lille 1, France)
Felipe Meneguzzi (CMU, USA)
John-Jules Meyer (Utrecht, Netherlands)
Frédéric Migeon (IRIT, Uni. Paul Sabatier, France)
Ambra Molesini (Bologna, Italy)
Pavlos Moraitis (Paris Descartes, France)
Haralambos Mouratidis (East London, UK)
Jörg Müller (Clausthal, Germany)
Peter Novak (Czech TU, Czech Republic)
Andrea Omicini (Bologna, Italy)
Julian Padget (Uni. Bath, UK)
Lin Padgham (RMIT, Australia)
Fabio Patrizi (Imperial college, UK)
Juan Pavon (Madrid, Spain)
Michal Pechoucek (Czech TU, Czech Republic)
Alexander Pokahr (Hamburg, Germany)
Enrico Pontelli (New Mexico state, USA)
Alessandro Ricci (Bologna, Italy)
Ralph Rönnquist (Intendico, Australia)
Chiaki Sakama (Wakayama Uni., Japan)
Valeria Seidita (Palermo, Italy)
Onn Shehory (IBM Haifa , Israel)
Guillermo Ricardo Simari (Uni Nacional del Sur, Argentina)
Tran Cao Son (New Mexico state, USA)
Nikolaos Spanoudakis (Tech. Uni. Crete, Greece)
Pankaj Telang (CISCO, USA)
Paolo Torroni (Bologna, Italy)
M. Birna van Riemsdijk (Delft, Netherlands)
Wamberto Vasconcelos (Aberdeen, UK)
Jørgen Villadsen (DTU Informatics, Denmark)
Gerhard Weiss (Maastricht, Netherlands)
Danny Weyns (Linnaeus, Sweden)
Wayne Wobcke (UNSW, Australia)
Pınar Yolum (Bogazici, Turkey)
Neil Yorke-Smith (American Uni Beirut / SRI, Lebanon / USA)
Additional Reviewers

Noroozian, Arman
Sabatucci, Luca
Program (Preliminary)

MONDAY 6TH MAY

Session 1: Commitments (9:15-10:30)

1. Welcome (9:15-9:30)
3. Matteo Baldoni, Cristina Baroglio and Federico Capuzzimati. 2COMM: a commitment-based MAS architecture

Coffee break (10:30-11:00)

Session 2: Design (11:00-12:30)

2. Yoosef Abushark and John Thangarajah. Propagating AUML Protocols to Detailed Design
3. Luca Sabatucci, Patrizia Ribino, Valeria Seidita, Carmelo Lodato, Salvatore Lopes and Massimo Cossentino. GoalSPEC: a Specification Language supporting Adativity and Evolution

Lunch (12:30-13:30)

Session 3: Infrastructure (13:30-15:30)

1. Rafael C. Cardoso, Jomi F. Hubner and Rafael H. Bordini. Benchmarking Communication in Actor- and Agent-Based Languages
2. Thiago Rodrigues, Antonio Carlos Rocha Costa and Graçaliz Dimuro. A Communication Infrastructure Based on Artifacts for the JaCaMo Platform

Coffee break (15:30-16:00)

Session 4: Keynote and discussion (16:00-17:30)
Session 5: Applications (9:00-10:30)


Coffee break (10:30-11:00)

Session 6: Conflicts and Decision making (11:00-12:30)

1. Akin Gunay and Pinar Yolum. Engineering Conflict-Free Multiagent Systems
2. Bernardo Luz, Felipe Meneguzzi and Rosa Vicari. Alternatives to Threshold-Based Desire Selection in Bayesian BDI Agents
3. Andreas Schmidt Jensen. Deciding Between Conflicting Influences

Lunch (12:30-13:30)

Session 7: Programming (13:30-15:30)

2. Alex Muscar. Agents for the 21st Century: the Blueprint Agent Programming Language
3. Mehdi Dastani and Marc van Zee. Belief Caching in 2APL
4. Sharmila Savarimuthu and Michael Winikoff. Mutation Operators for Cognitive Agent Programs

Coffee break (15:30-16:00)

Session 8: Panel (16:00-17:30)
Table of Contents (Papers are in the order of the presentation schedule)

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positron: Composing Commitment Protocols</td>
<td>1</td>
</tr>
<tr>
<td>Scott Gerard, Pankaj Telang, Anup Kalia and Munindar Singh</td>
<td></td>
</tr>
<tr>
<td>2COMM: a commitment-based MAS architecture</td>
<td>17</td>
</tr>
<tr>
<td>Matteo Baldoni, Cristina Baroglio and Federico Capuzzimati</td>
<td></td>
</tr>
<tr>
<td>Applying an O-MaSE Compliant Process to Develop a Holonic Multiagent System for the Evaluation of Intelligent Power Distribution Systems</td>
<td>33</td>
</tr>
<tr>
<td>Denise Case and Scott Deloach</td>
<td></td>
</tr>
<tr>
<td>Propagating AUML Protocols to Detailed Design</td>
<td>49</td>
</tr>
<tr>
<td>Yoosef Abushark and John Thangarajah</td>
<td></td>
</tr>
<tr>
<td>GoalSPEC: a Specification Language supporting Adativity and Evolution</td>
<td>65</td>
</tr>
<tr>
<td>Luca Sabatucci, Patrizia Ribino, Carmelo Lodato, Salvatore Lopes and Massimo Cossentino</td>
<td></td>
</tr>
<tr>
<td>Benchmarking Communication in Actor- and Agent-Based Languages</td>
<td>81</td>
</tr>
<tr>
<td>Rafael C. Cardoso, Jomi F. Hubner and Rafael H. Bordini</td>
<td></td>
</tr>
<tr>
<td>A Communication Infrastructure Based on Artifacts for the JaCaMo Platform</td>
<td>97</td>
</tr>
<tr>
<td>Thiago Rodrigues, Antonio Carlos Rocha Costa and Graçaliz Dimuro</td>
<td></td>
</tr>
<tr>
<td>Embedding agents in business applications using enterprise integration patterns</td>
<td>112</td>
</tr>
<tr>
<td>Stephen Cranefield and Surangika Ranathunga</td>
<td></td>
</tr>
<tr>
<td>An Infrastructure for the Design and Development of Open Interaction Systems</td>
<td>128</td>
</tr>
<tr>
<td>Daniel Okouya, Nicoletta Fornara and Marco Colombetti</td>
<td></td>
</tr>
<tr>
<td>Engineering Pervasive Multiagent Systems in SAPERE</td>
<td>144</td>
</tr>
<tr>
<td>Ambra Molesini, Andrea Omicini, Mirko Viroli and Franco Zambonelli</td>
<td></td>
</tr>
<tr>
<td>Engineering Industrial Multi-Agent Systems - The JIAC V Approach</td>
<td>160</td>
</tr>
<tr>
<td>Marco Luetzenberger, Tobias Küster, Thomas Konnerth, Alexander Thiele, Nils Masuch, Axel Hessler, Jan Keiser, Michael Burkhardt, Silvan Kaiser, Jakob Tonn and Sahin Albayrak</td>
<td></td>
</tr>
<tr>
<td>Co-ordinating Protocol Interleaving for Coherent Conversations</td>
<td>176</td>
</tr>
<tr>
<td>Wilson Wong, Lin Padgham, John Thangarajah and Wei Liu</td>
<td></td>
</tr>
<tr>
<td>Engineering Conflict-Free Multiagent Systems</td>
<td>192</td>
</tr>
<tr>
<td>Akin Gunay and Pinar Yolum</td>
<td></td>
</tr>
<tr>
<td>Alternatives to Threshold-Based Desire Selection in Bayesian BDI Agents</td>
<td>208</td>
</tr>
<tr>
<td>Bernardo Luz, Felipe Meneguzzi and Rosa Vicari</td>
<td></td>
</tr>
<tr>
<td>Deciding Between Conflicting Influences</td>
<td>224</td>
</tr>
<tr>
<td>Andreas Schmidt Jensen</td>
<td></td>
</tr>
<tr>
<td>Automatic BDI Plan Recognition from Process Execution Logs and Effect Logs</td>
<td>239</td>
</tr>
<tr>
<td>Hongyun Xu, Bastin Tony Roy Savarimuthal, Aditya Ghose, Evan Morrison, Qiying Cao and Youqun Shi</td>
<td></td>
</tr>
<tr>
<td>Agents for the 21st Century: the Blueprint Agent Programming Language</td>
<td>255</td>
</tr>
<tr>
<td>Alex Muscar</td>
<td></td>
</tr>
</tbody>
</table>
Belief Caching in 2APL ................................................................. 270
  Mehdi Dastani and Marc van Zee

Mutation Operators for Cognitive Agent Programs ................................. 286
  Sharmila Savarimuthu and Michael Winikoff
Abstract. We consider (commitment) protocols specified formally in terms of their roles, their messages, and the meanings of their messages (expressed as commitments). In an important advance over previous work, we show how to compose protocols, thereby facilitating reuse. We address two role-specific aspects of composition: (1) role requirements, capturing the benefits a role receives from the composite protocol; and (2) role accountability, capturing the commitments a role makes to other roles—to promote their joint enactments of the composite protocol. Our approach yields benefits in business requirements elicitation (natural abstraction); enactment (flexibility); and compliance and validation (ascribing accountability for each requirement to a specific role). We evaluate our contributions by modeling well-known protocols in the insurance, manufacturing, and healthcare domains.

Keywords: Commitments, Agent communication, Verification of multiagent systems, Communication protocols, Model checking

1 Introduction

We adopt an interaction-oriented stance on multiagent systems, e.g., as applied in cross-organizational service engagements. We consider (commitment) protocols, which specify the interactions between two or more roles in terms of how their messages relate to their commitments [24], obtaining well-recognized benefits in dealing with the autonomy and heterogeneity of business partners [2, 30].

Composition is a key construct in software engineering. We address two role-specific aspects of composition: role requirements which capture the benefits a role receives from the composite and role accountability which captures the commitments a role must make to other roles.

1.1 Real-Life Scenario: AGFIL

We illustrate our approach using the real-life, automobile insurance claims processing case for AGF Irish Life Holding (AGFIL) [3]. As Figure 1 illustrates, this case involves four parties along with POLICYHOLDER and ADJUSTER (not shown). AGFIL
underwrites automobile insurance policies and covers losses incurred by policy holders. Europ Assist (EA) provides a 24-hour help-line service for receiving claims. Approved REPAIRERS provide repair services. Lee Consulting Services (Lee) coordinates with AGFIL, repairers, and adjusters to handle a claim.

The traditional model of the AGFIL scenario describes the workflows of each partner along with how they relate to one another. Such a description, even if supported by standards such as BPMN [20], tightly couples the inner workings of the partners. Newer approaches deemphasize the inner workings and instead capture the interactions between the business partners more explicitly via a formal notation [12, 23, 28]. These approaches express constraints on the ordering and occurrence of the messages exchanged by the business partners.

In contrast, a commitment protocol emphasizes the social state of an interaction, expressed here in terms of commitments. A protocol describes the roles involved, the messages they exchange, and any preconditions on and effects of the messages on the social state. An agent adopts a role and enacts the specified protocol by autonomously choosing (in accordance with its internal policies) how to interact.

1.2 Contributions and Organization

Although protocols offer significant benefits over traditional approaches, protocols are not fully viable for the following reasons. One, specifying in one shot an adequate protocol for a complex scenario is nontrivial. Two, implementing agents who can play roles in such a comprehensive protocol is difficult because the differing details of the protocols complicate reusing parts of agent implementations. Our contribution in this paper is to show how complex protocols can be constructed by composing existing protocols. Previous relevant research falls into these categories: (a) commitments but not composition [10]; (b) composition but no commitments [19, 25]; and (c) composition and...
commitments. The last category can be categorized as (c1) purely abstract description without a specification language or tools [15]; (c2) composition of commitment-based protocols based on regulative constraints [16]; and (c3) our approach to composition of commitment-based protocols based on role responsibilities and accountabilities.

Our approach, *Positron*, extends the Proton approach [10] to provide a clear syntax and semantics for composite protocols. Where Proton checks protocol refinement, Positron composes protocols. Positron (a) recursively expands nested constituent protocols; (b) introduces *composite protocol diagrams* as a graphical notation, conveying important features of the composite protocol to business and technical stakeholders; (c) introduces *role requirements* and *role accountabilities*; (d) incorporates a methodology for composing commitment protocols; and (e) implements a decision procedure and mechanical verification of protocols with respect to role requirements, role accountabilities, and enactments, compiling formulas to temporal logic, and employing MCMAS [13], a leading model checker, to verify if the composite protocol satisfies those formulas.

We describe relevant background, the major elements of our technical approach, and our methodology for constructing composite protocols. We evaluate our approach by modeling protocols from the insurance, manufacturing, and healthcare domains, that other researchers have studied, and summarize our results and experiences. We conclude with a discussion of the relevant literature and future work.

2 Background

We formulate a commitment [24] as being *from* a set of debtors *to* a set of creditors that if the antecedent begins to hold, the debtors will bring about the consequent in the future. In symbols: $C_{\{\text{debtors}\},\{\text{creditors}\}}(\text{antecedent, consequent})$ [10]. Both antecedent and consequent are Boolean expressions over state-space propositions. When the antecedent becomes true, the commitment is *detached*, and the debtors become *unconditionally* committed to the creditors. When the consequent becomes true, the commitment is *discharged*. Debtors *should* discharge their detached commitments. However, debtors are autonomous and may violate a commitment by canceling it. The only compliance requirement for commitments is: each detached commitment *must* eventually be discharged (satisfied, transferred or released) or canceled. Specifically, there is no implicit ordering constraint between the antecedent and consequent.

3 Technical Approach

Positron provides a formal language in which to express composite protocols based on existing constituent protocols. (We lack the space to present the detailed syntax, but present an example in Listing 1 later.) Recall that Proton provides a language for capturing roles, propositions, and messages with guards on when they can be sent, and their effects on the commitments of roles [10]. Positron augments the Proton language by adding constructs to define a composite protocol *using* a set of parameterized constituent protocols and defines a protocol composition methodology.
Further, while it accepts and verifies any CTL expression, Positron introduces five
constructs for common verification patterns when composing protocols: \textit{Req} function
for role requirements, \textit{coupling commitments} for role accountabilities, and three \textit{path
expressions} for good and bad enactments.

3.1 Protocol Composition

Positron supports nested composition of protocols. A composite protocol $P$ can use
(include) an instance of a constituent protocol with a \texttt{use} statement $q : Q(x = \bar{p})$
specifying instance name ($q$), protocol type ($Q$), a set of arguments $\bar{p}$ passed by $P$, and
a matching set a parameters $x$ accepted by constituent $Q$. Arguments and parameters
are named and have a type of either role or proposition. The argument and parameter
sets must contain matching names and types. Positron expands $P$ to produce a new,
flatter protocol $P'$. Expansion gives every element in $Q$ a new, unique name, and re-
places each parameter with its corresponding argument. Unique names are constructed
by prepending the instance name $q$ to each element name in $Q$.

\textbf{Definition 1.} Given a set of arguments $\bar{p}$, a parameterized constituent protocol instance
$Q$ that accepts a set of parameters $\bar{x}$, and the sets $\bar{p}$ and $\bar{x}$ agree in both name and type.
Define $Q_{\bar{p}}^{\bar{x}}$ as $Q$ in which all elements in $Q$ are given unique names, and every parameter
in $\bar{x}$ is replaced with its corresponding argument in $\bar{p}$.

Protocol expansion of a composite $P$ containing multiple constituent instances $q : Q(x = \bar{p})$ is the union of $P$ and $Q_{\bar{p}}^{\bar{x}}$, and removing the use statement $Q_{\bar{p}}^{\bar{x}}$ from $P$’s uses.
The definition expands any one constituent. Apply the definition repeatedly to expand
all constituent instances.

\textbf{Definition 2.} Given a composite protocol $P$ that uses multiple constituent protocol in-
stances $q : Q(x = \bar{p})$, where $P$ passes a set of arguments $\bar{p}$, $Q$ accepts a set of parameters $\bar{x}$, and the sets $\bar{p}$ and $\bar{x}$ agree in name and type. Then protocol
$P' = \text{expand}(P, q : Q(x = \bar{p})$) is the expanded version of $P$ and $Q$, and is defined as

\begin{align*}
\text{roles}(P') & := \text{roles}(P) \cup \text{role}(Q_{\bar{p}}^{\bar{x}}) \\
\text{props}(P') & := \text{props}(P) \cup \text{props}(Q_{\bar{p}}^{\bar{x}}) \\
\text{commitments}(P') & := \text{commitments}(P) \cup \text{commitments}(Q_{\bar{p}}^{\bar{x}}) \\
\text{messages}(P') & := \text{messages}(P) \cup \text{messages}(Q_{\bar{p}}^{\bar{x}}) \\
\text{checks}(P') & := \text{checks}(P) \cup \text{checks}(Q_{\bar{p}}^{\bar{x}}) \\
\text{uses}(P') & := (\text{uses}(P) - q) \cup \text{uses}(Q_{\bar{p}}^{\bar{x}})
\end{align*}

where \text{roles}(Q), \text{props}(Q), \text{commitments}(Q), \text{messages}(Q), \text{checks}(Q), and \text{uses}(Q)
refer to the corresponding element sets of protocol $Q$.

3.2 Role Requirements

\textit{Role requirements} are the requirements that an agent playing a role places on the com-
posite protocol. A designer specifies a role requirement in the Positron language, which
Positron compiles into a CTL formula. As an example, in the AGFIL scenario, POLICY-HOLDER expect his car will be repaired if he has an accident. Positron could compile this requirement into the CTL specification: $AG(accident \implies AF \text{repair})$.

However, such a requirement ignores business exceptions: a commitment may fail because its debtor either chooses not to, or is prevented by circumstances from, discharging it. In verifying a role requirement, we cannot assume commitments are never canceled. Rather, we state role R’s requirement as: if R fulfills all its commitments and $p$ holds at any state, then always eventually, either $q$ holds or a role other than $R$ cancels one of its commitments. If $R$’s requirement fails because $R$ cancels a commitment, that is not a fault of the protocol, but of $R$. In CTL, where $r.anyCancel$ is true if and only if role $r$ cancels any of its commitments, this is

$$Req(R, p, q) \triangleq AG(p \implies AF (q \lor \bigvee_{r \neq R} r.anyCancel))$$

In AGFIL, one of POLICY-HOLDER’s role requirements is captured as: if INSURER offers coverage, I paid the premium, and I have an accident, then my car will be repaired: $Req(\text{PH}, \text{coverage} \land \text{premium} \land \text{accident, repair})$

### 3.3 Enactment Requirements

Although capturing all possible enactments is not feasible, designers often know of specific good and bad enactments. We use the specified enactments as bases for verifying a composite protocol to assist designers in refining the protocol specification (e.g., its constituent protocols and coupling commitments) or the requirements. Our notion of enactments resembles scenarios from scenario-based requirements engineering [9]. In essence, each enactment corresponds to a unit test in software engineering.

We use model checking to verify enactments. We introduce three recursive functions to simplify enactment specification. Given an enactment $P$, which is an ordered list of Boolean expressions over states and messages, where head($P$) is the first element in list $P$, and tail($P$) is $P$ without the first element, let

$$EXPath(P) \triangleq \begin{cases} head(P) \land EX(EXPath(tail(P))) & \text{if } |P| > 1 \\ \top & \text{if } |P| = 1 \end{cases}$$

$$EFPath(P) \triangleq \begin{cases} EF(head(P) \land EFPath(tail(P))) & \text{if } |P| > 1 \\ \top & \text{if } |P| = 1 \end{cases}$$

$$EUPath(r, P) \triangleq \begin{cases} E(\neg r \lor (head(P) \land EUPath(r, tail(P)))) & \text{if } |P| > 1 \\ E(\neg r \lor P) & \text{if } |P| = 1 \end{cases}$$

$EXPath$: specifies a path of states, beginning at a start state, that must appear consecutively without skipping over other states. In protocols with many constituents, $EXPath$ can be too strong a constraint, since it precludes interleaving of constituent protocols.

$EFPath$: specifies a path of states that must appear in order, but allows other states to be interleaved. This is a weaker constraint than $EXPath$.

$EUPath$: specifies a path of states that must appear in order, and constrains which states can be interleaved in the path. Expression $r$ identifies which states must not be
interleaved in the path. An \textit{EUPath} constraint is stronger than \textit{EFPath} and weaker than \textit{EXPath}.

Two enactments from AGFIL are

\[
\text{EFPath}(\text{accident, deliverReq, deliverCar, \ldots, repair})
\]
\[
\neg \text{EFPath}(\text{repair, accident})
\]

3.4 Coupling Commitments

The constituent protocols occurring in a (nontrivial) composite protocol must be interrelated. In a multiagent system, some role must be accountable for ensuring constituent protocols are properly interrelated. We capture the \textit{role accountability} implied by such an interrelationship via a coupling commitment. A coupling commitment’s debtor is the accountable role, and its creditors are (in general) the union of all roles connected by the interrelated constituent protocols, minus the debtor. Like any commitment, debtor commits to discharge consequent if antecedent becomes true.

\[C_{\text{accountable role}, \{\text{interrelated roles}\}}(\text{antecedent, consequent})\]

Two coupling commitments from AGFIL are

\[C_{\text{CC}, \{\text{PH, Re}\}}(\text{deliverReq, notifyRE})\]
\[C_{\text{PH}, \{\text{CC, Re}\}}(\text{deliverReq } \land \text{ approval, deliverCar})\]

3.5 Verification

Positron reads designer written source code for the composite and constituent protocols and generates a single MCMAS input file. MCMAS reads the input, builds the model, and reports whether each CTL formula holds in the model.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig2.png}
\caption{Selected states and transitions for AGFIL.}
\end{figure}

Figure 2 shows a portion of the state space Positron generates for verification from AGFIL’s constituent protocols and coupling commitments. The start state is denoted by the unlabeled line in the top left. Solid black lines are valid transitions (messages);
dashed red lines are invalid transitions. Since the message guard for coverage is premium, coverage can occur only after premium, making $s_1$ an invalid start state. The other states are also invalid start states.

Notice that the top row (premium, coverage, accident, and repair) begins a good enactment. Positron can verify the existence of this path using $\textit{EFPath}$ requirement. Further, Positron can ensure that the model is free of specific bad enactments, for example, that $s_1$ must not be a start state.

To capture the only commitment compliance requirement, Positron generates a model checking fairness constraint for each commitment: a commitment must not remain unconditional and unresolved forever. Red dashed loops are invalid because they violate a fairness constraint.

A composite protocol may fail to satisfy role or enactment requirements for different reasons.

$\text{Ver}_{RR}$: If a role requirement formula based on the Req function fails, then coupling commitments are missing; add coupling commitments that require agents to act.

$\text{Ver}_{G}$: If a good enactment formula fails, then either ($\text{Ver}_{GG}$) some message guards are too strong; weaken guards to validate additional good transitions. Or ($\text{Ver}_{GC}$) some commitment become detached, but can never resolve; weaken guards to validate transitions that satisfy the commitment’s consequent.

$\text{Ver}_{B}$: If a bad enactment formula fails, then some message guards are too weak; strengthen guards to invalidate existing incorrect transitions.

3.6 Composite Protocol Diagrams

We propose composite protocol diagrams (CPDs) as a notation that displays the essence of a composite protocol both to business analysts and technical designers, who collaborate in its construction. We use CPDs in this paper to help visualize a large amount of protocol information, but we defer evaluation of this visual notation to future work. CPD diagrams focus designers’ attention by summarizing high-level business relationships between the roles as reusable constituent protocols, hiding the details of each constituent.

Figure 3 shows the CPD for composite protocol AGFIL. It contains constituent protocol instance PH-IN of type $\textit{Exchange}$. PH $R1$ PH-IN shows role PH enacts role $R1$ in constituent protocol PH-IN. The two unlabeled circular arcs centered on POLICYHOLDER represent coupling commitments, implicitly named $PH_1$ (inner) and $PH_2$ (outer). (role names REQ and R1 label straight edges, not arcs.)

4 Methodology

This section describes, and Table 1 summarizes, our iterative methodology to develop a CPD such as that of Figure 3.

$\text{Step 1 (Roles)}$: Identify all roles with a business function, i.e., that potentially enter into commitments with others.

$\text{Step 2 (Constituent Selection)}$: Identify all business relationships among different subsets of roles and identify constituent protocols that realize such relationships.
Step 3 (Role Requirements): Specify each role’s requirements as Req functions.

Step 4 (Enactments): Incrementally specify good and bad enactments. The enactments can be developed by a role-playing process similar to that described by Parunak [22]. They should cover representative enactments that support or invalidate each of the role requirements identified in the previous step. Manually tracing enactments throughout a model is tedious and error prone. Explicitly specifying enactments enables the model checker to trace the enactments mechanically, after each incremental composite change.

Step 5 (Coupling Commitments): Examine each good enactment from beginning to end, and assume roles do only what is minimally required to discharge their com-
Table 1. Inputs and outputs for each step of the methodology.

<table>
<thead>
<tr>
<th>Step Name</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Roles</td>
<td>Background and requirements</td>
<td>Roles in the composite</td>
</tr>
<tr>
<td>2 Constituent Selection</td>
<td>Role relationships and protocol library to be composed</td>
<td>Constituent protocols</td>
</tr>
<tr>
<td>3 Role Requirements</td>
<td>Role business needs</td>
<td>Role requirements</td>
</tr>
<tr>
<td>4 Enactments</td>
<td>Background knowledge of requirements</td>
<td>Good and bad enactments</td>
</tr>
<tr>
<td>5 Coupling Commitments</td>
<td>Enactments complete CPD</td>
<td>Coupling commitments;</td>
</tr>
<tr>
<td>6 Positron</td>
<td>All artifacts</td>
<td>Positron source code</td>
</tr>
<tr>
<td>7 Verification</td>
<td>Positron source code</td>
<td>Model checker results</td>
</tr>
</tbody>
</table>

We evaluate our contributions by modeling protocols from the insurance, manufacturing, and healthcare domains.

5 Evaluation

We evaluate our contributions by modeling protocols from the insurance, manufacturing, and healthcare domains.

5.1 AGFIL Evaluation

We extend the AGFIL protocol by adding (a) POLICYHOLDER and accident reporting; (b) ADJUSTER and the redirection of two messages between CLAIMHANDLER and REPAIRER through ADJUSTER; (c) payments from REPAIRER to CLAIMHANDLER to INSURER; (d) a protocol for premiums and coverage between POLICYHOLDER and INSURER; and (e) REPAIRER returning the car.

At the top of Figure 3, POLICYHOLDER (enacting role R1) purchases insurance from INSURER (enacting role R2) using an instance of constituent protocol Exchange named PH-IN. POLICYHOLDER reports accidents to CALLCENTER using an instance of constituent protocol RequestResponse named PH-CC. CALLCENTER notifies INSURER (IN-CC), and assigns and notifies REPAIRER (CC-RE). INSURER passes the claim to CLAIMHANDLER (IN-CH). POLICYHOLDER and REPAIRER exchange the damaged and later repaired car (PH-RE). CLAIMHANDLER, REPAIRER, and ADJUSTER inspect the car and approve repairs (CH-AD, RE-AD and CH-Re).
**Step 1 (Roles):** Figure 1 refers to specific agents (companies), not roles. Declare six roles: role **INsurer** (abbreviated **IN**) for agent AGFIL, **CALLCenter** (CC) for EUROP ASSIST, and **CLAIMHandler** (CH) for LEE. The other roles are **POLICYHolder** (PH), **REPAIRer** (RE), and **ADJUSTER** (AD). At the end of this step, the CPD diagram in Figure 3 shows only the six shaded role nodes.

**Step 2 (Constituent Selection):** Assume protocols RequestResponse and Exchange (where two roles swap items) already exist. Designers recursively create **Claims** for IN-CH and **ApprovedWork** for CH-RE. Designers add the constituent protocol nodes and edges to Figure 3, completing all CPD nodes and edges.

**Step 3 (Role Requirements):** **POLICYHolder** requires: (1) if he has coverage, pays his premium, and has an accident, his car is repaired; (2) if he delivers his car to **REPAIRer**, his car is returned. **INsurer** requires: if a claim is filed, the claim is finalized. All roles except **POLICYHolder** require payment if they perform their tasks. All these are described as Req functions. Role requirements can also be specified directly in CTL, e.g., **INsurer** requires no car repairs without an inspection: $AG(\neg(repair \land \neg inspectCH))$.

**Step 4 (Enactments):** An important good enactment is that of reporting an accident and getting car repaired: (a) **POLICYHolder** reports an accident to **CALLCenter** (PH-CC); (b) **CALLCenter** assigns and notifies **REPAIRer** to repair the car (CC-RE); (c) **CALLCenter** asks **POLICYHolder** to deliver his car to a specific **REPAIRer** (PH-CC); (d) **POLICYHolder** delivers car to **REPAIRer** (PH-RE); Remaining steps are omitted. Performing repairs before an accident is reported is a bad enactment: (e) car repaired; (f) accident reported.

**Step 5 (Coupling Commitments):** Between messages (a) and (b) of the accident-reporting enactment (see previous step), if **POLICYHolder** reports an accident, **CALLCenter** assigns and notifies **REPAIRer**: $C_{CC,\{PH,RE\}}(accident, notifyRe)$ Between messages (c) and (d), if **CALLCenter** asks **POLICYHolder** to deliver his car to **REPAIRer**, he does so.

\[ PH_1 = C_{PH,\{CC,RE\}}(deliverReq \land approval, deliverCar) \]

Adding arcs, the complete AGFIL CPD is Figure 3.

**Step 6 (Positron):** Listing 1 shows some lines from the Positron source file for AGFIL. Lines 2-3 declares all roles and propositions. Line 4 instantiates an instance of **Claims** named IN-CH. Lines 10 and 11 are two coupling commitments. Line 14 lists one of **POLICYHolder**’s role requirements. Line 15 lists an **INsurer** requirement as explicit CTL. Line 17 verifies the good, accident-reporting enactment that must exist in the composite, and Line 18 verifies a bad enactment that must not exist.

The conversion of Positron to MCMAS maps role to role; proposition to boolean; commitment to enum and fairness condition; message to action; and formula expansion to formula. It also defines proposition and commitment evolutions, maps high-level Positron expressions to low-level MCMAS expressions, tracks each agent’s last action and anyCancel values, and generates a large number of proposition and commitment state evaluation statements.

**Step 7 (Verification):** Running Positron and MCMAS successfully verified nine CTL formula: eight role and one enactment requirements. Removing any single coupling commitment caused one or more formulas to fail.
6 Results and Experience

To demonstrate the broad applicability of Positron, our methodology was successfully able to create composite protocols for scenarios from three different business domains: AGFIL from insurance; Quote To Cash, an important business process for manufacturing supply chains [21]; and ASPE, a healthcare process for breast cancer diagnosis [1]. Positron successfully verified all role and enactment requirements. Table 2 shows statistics and timings for these three protocols.

**Model Verification:** We encountered and fixed a number of verification failures. Good enactment failures (VerGG and VerGC) were generally easier to fix, since they only require that some path exist. Bad enactment failures (VerB) require the impossibility of a particular enactment.

**Model Validation:** Identifying requirements was straightforward, but some initial specifications were incorrect because preconditions were missed. For example, POLICYHOLDER’s role requirement on Line 14 initially failed (VerRR) because coupling commitment PH1 on Line 11 did not include approval; REPAIRER will not always repair a car just because it is delivered to him. And, an accident is insufficient to get POLICYHOLDER’s car is repaired; POLICYHOLDER must also have a policy and pay the premium.

Positron generates model checking fairness conditions to ensure all unconditional commitments eventually resolve. Initially, AGFIL’s good enactment on Line 17 mysteriously failed because, even though the model allowed all the transitions, the good enactment had unresolvable commitments (invalid by commitment fairness conditions). We corrected the model so all commitments could resolve.
Table 2. Positron statistics. (M is $10^6$ and G is $10^9$.)

<table>
<thead>
<tr>
<th>Composite Metric</th>
<th>AGFIL</th>
<th>QTC</th>
<th>ASPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constituent instances</td>
<td>11</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Roles</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Propositions</td>
<td>22</td>
<td>37</td>
<td>18</td>
</tr>
<tr>
<td>Commitments (all)</td>
<td>24</td>
<td>43</td>
<td>12</td>
</tr>
<tr>
<td>Coupling commitments</td>
<td>9</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>Messages</td>
<td>22</td>
<td>55</td>
<td>20</td>
</tr>
<tr>
<td>CTL Formulas</td>
<td>9</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>Role requirements</td>
<td>8</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>Enactment requirements</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Positron statements</td>
<td>94</td>
<td>164</td>
<td>81</td>
</tr>
<tr>
<td>State space size</td>
<td>120M</td>
<td>381G</td>
<td>1.47M</td>
</tr>
<tr>
<td>Positron processing time</td>
<td>1.98s</td>
<td>3.16s</td>
<td>1.68s</td>
</tr>
<tr>
<td>MCMAS processing time</td>
<td>4.29s</td>
<td>1274s</td>
<td>5.78s</td>
</tr>
<tr>
<td>Total time</td>
<td>6.27s</td>
<td>1278s</td>
<td>7.46s</td>
</tr>
</tbody>
</table>

7 Discussion

Positron gains an advantage over traditional approaches by focusing on high-level business relationships realized as constituent protocols, and by focusing on commitments rather than control flow. Because role accountabilities are stated as commitments, if a requirement fails, we can trace the failure back to a specific role.

CPDs summarize relevant details about a composite protocol and we expect they will prove valuable, because they bring together both technical and business descriptions of protocols, helping bridge the Business-IT Divide [26].

7.1 Relevant Literature

Table 3 compares Positron with other work. Some papers propose a protocol specification language, and some propose an accompanying protocol specification methodology. Some papers address single protocols in isolation; some address common patterns within protocols; some address the composition of multiple protocols to create new composite protocols. Of those papers that address verification, some address business level requirements; some address verification properties between two protocols or models (such as protocol refinement); some address protocol-wide properties; some verify properties that must hold between the constituents of a composite protocol; some formulate role-specific properties; some formulate good or bad enactment properties; and some address other verification topics not addressed above.

Desai et al. [6] propose OWL-P [7, 5] and MAD-P [8] for specifying and verifying commitment protocols and their compositions. They employ axioms to specify a composition. These approaches suffer from a key drawback: axiom violations are not assigned to any particular role. In contrast, Positron employs coupling commitments with clear role accountability for the effects of one constituent protocol on others. Further, Amoeba is purely manual, whereas Positron incorporates mechanical verification.
Table 3. Approach comparison. Column abbreviations and citations are Po=Positron; Pr=Proton; DA, DO, Dv and DM=Desai et al.; T=Telang and Singh; Y=Yolum; Mi=Miller and McBurney; G=Günay et al.; C=Cheong and Winikoff, Mc=McGinnis and Robertson, L=Lomuscio et al., Ma=Marengo. Check marks show the significant topics addressed by each paper. The cell contents of the verification rows indicates whether the paper discusses (D) or mechanizes (M) verification.

<table>
<thead>
<tr>
<th>Significant Topics Verification Topics</th>
<th>Po</th>
<th>Pr</th>
<th>DA</th>
<th>DO</th>
<th>Dv</th>
<th>DM</th>
<th>T</th>
<th>Y</th>
<th>Mi</th>
<th>G</th>
<th>C</th>
<th>Mc</th>
<th>L</th>
<th>Ma</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocol specification</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Methodology</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Single protocol</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Protocol patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Protocol composition</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Requirements verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Business</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Protocol-to-protocol</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Protocol</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Inter-constituent</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Role-specific</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Enactments</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Other</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>

Adopting Amoeba’s event ordering idea would add flexibility to our approach, but more granular parameterizations of constituents provides the same functionality.

Telang and Singh [27] (T&S) describe a methodology for business modeling that captures the commitments to be created among the parties by melding selected business patterns. In contrast, a protocol in Positron additionally specifies the messages and guards, and the protocols are first-class entities that retain their identity in the composite protocol, yielding improved modularity and modifiability. Most significantly, T&S’s approach verifies if one implementation is sound with respect to the model. In contrast, Positron verifies if the model itself is sound.

Yolum [29] proposes generic correctness properties of commitment protocols for design-time verification, but does not address composite protocols. She considers generic properties, whereas we consider role-specific business requirements. It would be interesting to formulate Yolum’s generic correctness properties in Positron.

Miller and McBurney [18] (M&M) propose the $\mathcal{RAS.A}$ language based on propositional dynamic logic (PDL) to specify and compose protocols. $\mathcal{RAS.A}$’s preconditions, actions and postconditions correspond to Positron’s guards, messages and meanings. Positron additionally incorporates role requirements, coupling commitments, and good and bad enactment paths, making Positron practically viable (intentionally omitted from $\mathcal{RAS.A}$). Theses are important in naturally describing business protocols, as we demonstrated above. Whereas M&M describe a custom reasoner, we rely on CTL semantics as realized in MCMAS.
Günay et al. [11] treat protocols as sets of commitments and propose automatically generating such sets from an agent’s beliefs, goals, and capabilities. In contrast, we offer a semiautomatic approach where a tool helps designers compose existing protocols. Automatic generation is attractive but may not be feasible for complex settings, although a hybrid approach of developing atomic protocols mechanically and composite protocols with human assistance might be viable.

Cheong and Winikoff [4] describe the Hermes system for goal-oriented interaction. They focus on interaction-level goals, where we focus on role-level requirements and commitments. Their action sequence diagrams capture only good enactments.

McGinnis and Robertson [17] propose an approach in which an agent sends a protocol specification to other agents at runtime, as a way to accomplish dynamic, runtime, protocol adaptation. They remark that their approach lacks a way to prevent the agents from making an undesirable change to a protocol. If their protocols were augmented with commitments, Positron can help address this gap. For example, an agent may not remove a message from a protocol that brings about the consequent of a detached commitment. Why they describe rules for dynamically changing protocols, they do not address formal verification of interaction properties.

Lomuscio et al. [14] semiautomatically compile and verify contract-regulated service compositions. They use temporal-epistemic logic to check whether agents comply with their contracts using MCMAS (the same tool we employ). A crucial difference is Lomuscio et al. consider service compositions; we consider protocol compositions. Since a protocol has a distributed footprint, their compositions are inherently more subtle. Table 3 classifies them as composition, and includes verification between actual behaviors and contractually correct behaviors (two “protocols”).

Marengo [16] considers a related problem where protocols are composed (grafted) using regulative specifications (constraints) and LTL. We propose a methodology and use role responsibilities and role accountabilities using CTL. These idea sets are complementary and worthy of further study.

BPMN [20] is an industry standard notation for business processes. Unlike Positron, BPMN is only semiformal, does not lend itself to formal verification, and emphasizes control flow in rigid processes. Positron minimally constrains the participants by specifying the process in terms of commitments. Protocols are building blocks in Positron.

7.2 Future Work

There are many useful future directions. At the theoretical level, treating the goals of the participants [11] is natural. At the practical level, generating enactments via tooling would be valuable. At the empirical level, evaluating the effectiveness of Positron (the approach and the tool) with professional developers on cross-organizational business processes would be necessary to promote the adoption by industry.
References


2COMM: a commitment-based MAS architecture

Matteo Baldoni, Cristina Baroglio, Federico Capuzzimati

Università degli Studi di Torino — Dipartimento di Informatica
c.so Svizzera 185, I-10149 Torino (Italy)
{matteo.baldoni,cristina.baroglio,federico.capuzzimati}@unito.it

Abstract. This paper proposes a multi-agent system architecture where agent interaction is ruled with the help of commitment-based interaction protocols. Commitment protocols are embodied into artifacts which can be accessed and used by the interacting agents. Ideally, the architecture is orthogonal to the language that is used to specify the commitment protocols. In this paper we rely on Yolum and Singh’s proposal. The implementation that is described relies on the well-known JADE and CArtAgO frameworks.

Keywords: Commitment, Commitment-based Interaction Protocols, Agents & Artifacts Model, JADE, JADE Methodology, Agent-Oriented Software Engineering

1 Introduction and Motivation

Interaction creates social expectations and dependencies in the involved partners [27, 12, 24, 17]. These should be explicitly accounted for by the agent platform to allow the coordination of autonomous entities. In order to create social expectations on the agents’ behavior, it is necessary to introduce a normative characterization of coordination and give a social meaning to their actions. An agent that understands such a specification and that publicly accepts it (i.e. that declares it will behave according to it) allows reasoning about its behavior [15]. This is the key to the development of open environment systems, made of autonomous and heterogeneous components. By not supplying such abstractions, current platforms do not supply agents the means for observing or reasoning about such meanings of interaction, and do not supply the designers the means to explicitly express and characterize them when developing an interaction model.

The aim of this work is to fill the gap by introducing in JADE the means for exploiting commitments and commitment-based protocols, which are well-known for featuring the social and observational semantics [24, 25, 30], JADE notoriously lacks of. Following [3], we perform such an extension by enabling a form of indirect communication among agents with the help of artifacts: commitment-based communication artifacts implement interaction protocols as well as monitoring functionalities for the verification that the on-going interaction respects
the protocol, for detecting violations and violators, and so forth. Artifacts, therefore, encode the social layer of the multi-agent system: as a programmable communication channel an artifact contains what in the terminology of commitment protocols is called “the social state”, and captures it as an interaction session among the parties. Artifacts also supply agents the social actions that are necessary to the interaction – that is, actions that allow agents to enter into and to comply with commitments – together with their social meaning and, as a consequence, they capture the coordination rules of the protocol. The reification of commitment protocols allows agents to act on them, e.g. to examine them (for instance, to decide whether to play one of the foreseen roles), use them (which entails that they explicitly accept the corresponding regulation), negotiate their construction, specialize them, and compose them. The advantage of relying on indirect communication is that it allows more variegated ways of interacting, not hindering message exchange when necessary.

JADE [4], [5] is a well-established development environment for multi-agent systems. It is FIPA-compliant and actually used for industrial applications. Our starting point for introducing commitment-based protocols inside JADE was the JADE Methodology [22]. This methodology is particularly interesting because it is intrinsically agent-oriented and it is not the adaptation of an object-oriented methodology, and it combines a top-down approach with a bottom-up one, allowing integration with eventually current legacy, non agent-based systems. It concerns two of the four main phases of the standard software development cycle: the analysis phase and the design phase. Our proposal can be integrated seamlessly within the JADE Methodology, simply by substituting the selection of JADE FIPA protocols with the selection/construction of appropriate communication artifacts. We also use the methodology to show the differences between these two alternatives with the help of an example from a financial setting.

Section 2 reports the relevant background, necessary to understand the proposal. Section 3 is the core of the paper, containing the original proposal. Section 4 applies the concepts to an illustrative example, from a financial setting. A discussion also involving related works ends the paper.

2 Background

We briefly report the technical, methodological and theoretical background required for our work. We use the proposal in [2] as a high-level reference architecture. In this work, the authors outline the basic ideas for an interaction-oriented agent framework, grounding the social semantics of interaction on commitments, and proposing the A&A (Agents and Artifacts) Metamodel as a means to obtain a form of indirect, observable communication. Let us, then, explain the fundamental bricks to build our architecture, whose overview is reported in Fig. 1.

*JADE framework.* JADE is a popular and industry adopted agent framework. It offers to developers a Java middleware 100% FIPA-compliant (Foundation for Intelligent Physical Agents, [1]) plus a set of command-line and graphical
tools, supporting development and debugging/testing activities. Its robustness and well-proven reliability makes JADE a preferred choice in developing MAS. It is currently used in many research and industrial projects jointly with its most popular and promising extension, WADE [11].

A JADE-based system is composed of one or more containers, each grouping a set of agents in a logical node and representing a single JADE runtime. The overall set of containers is called a platform, and can spread across various physical hosts. The resulting architecture hides the underlying layer, allowing support for different low-level frameworks (JEE, JSE, JME, etc.). The platform reference container is called main container, and represents the entry point to the system. JADE provides communication and infrastructure services, allowing agents, deployed in different containers, to discover and interact with each other, in a transparent way from the developer’s logical point of view.

**Commitment Protocols.** Agents share a social state that contains commitments and other literals that are relevant to their interaction. A commitment \( C(x, y, r, p) \) denotes a contractual relationship between a debtor \( x \) and a creditor \( y \): \( x \) commits to \( y \) to bring about the consequent condition \( p \) when the antecedent condition \( r \) holds. A commitment, when active, functions as a directed obligation from a debtor to a creditor. However, unlike a traditional obligation, a commitment may be manipulated, e.g., delegated, assigned, or released [26]. Importantly, commitments have a regulative value: the social expectation is that agents respect the commitments which involve them and, in particular, the debtor is considered responsible of realizing the consequent condition. Thus, the agents’ behavior is affected by the commitments that are present in the social state. A commitment protocol usually consists of a set of actions, whose semantics is shared (and agreed upon) by all of the interacting agents [30, 29, 14]. The semantics of the social actions is given in terms of operations which modify the social state by, e.g., adding a new commitment, releasing another agent from some commitment, satisfying a commitment, see [30].

**CArtAgO.** CArtAgO is a framework based on the A&A model. It extends the agent programming paradigm with the first-class entity of artifact: a resource that an agent can use, and that models working environments ([23]). In order to properly model a MAS, CArtAgO proposes to explicitly model the environment where pro-active agents live, work, act and communicate. It provides a way to define and organize workspaces, logical groups of artifacts, that can be joined by agents at runtime and where agents can create, use, share and compose artifacts to support individual and collective, cooperative or antagonistic activities. The environment is itself programmable as a dynamic first class abstraction, it is an active part of a MAS, encapsulating services and functionalities. The A&A model decouples the notion of agent from the notion of environment. The overall engineering of the MAS results more flexible, easy to understand, modular and reusable.

CArtAgO provides an API to program artifacts that agents can use, regardless of the agent programming language or the agent framework used. This is
possible by means of the agent body metaphor: CArtAgO provides a native agent entity, which allows using the framework as a complete MAS platform as well as it allows mapping the agents of some platform onto the CArtAgO agents, which, in this way, becomes a kind of “proxy” in the artifacts workspace. The developed agent is the mind, that uses the CArtAgO agent as a body, interacting with artifacts and sensing the environment. An agent interacts with an artifact by means of public operations. An operation can be equipped with a guard: a condition that must hold so that the operation will produce its effects. It is not an execution condition: when the guard does not hold the action is performed anyhow but without consequences.

Artifacts naturally lend themselves to provide a suitable means for realizing mediated communication channels among agents. To this aim, it is necessary to encode inside the communication artifacts a normative characterization to the actions it offers to agents and that allow them to interact. We propose to interpret commitment protocols as environments, within which agents interact. The public interface of artifacts allows agents to examine the encoded interaction protocol. As a consequence, the act of using an artifact can be interpreted as a declaration of acceptance of the coordination rules. This will generate social expectations about the agent’s behavior and agrees with the characterization of norms in [15]. Moreover, the fact that the behavior of agents on artifacts is observable and that interactions only occur through artifacts, agrees with the view that regulations can only concern observable behavior [16]. The resulting programmable environment provides a flexible communication channel that is suitable for realizing open systems. Notice that the use of a programming environment does not mean that the social state will necessarily be centralized: an artifact can be composed by a distributed network of artifacts.

3 2COMM: Reifying Commitment Protocols with Artifacts

In this section, we describe the original contribution of this work, which is the realization of a commitment-based MAS architecture that we named 2COMM (Communication & Commitment). To this aim, we rely upon the JADE and the CArtAgO frameworks, introducing a mediated form of interaction among JADE agents. We realized mediated interaction by means of communication artifacts, which, in our proposal, replace the JADE-based FIPA protocols and which reify commitment-based protocols [3]. In Fig. 1 we draw the basic architecture of 2COMM. At the bottom level, the JADE framework supplies standard agent services: message passing, distributed containers, naming and yellow pages services, agent mobility. When needed, an agent can enact a certain protocol role, thus using a communication artifact by CArtAgO. This provides a set of operations by means of which agents participate in a mediated interaction session. Each artifact (protocol enactment) maintains a social state, that is, a collection of social facts and commitments involving the roles of the corresponding protocol, following Yolum and Singh’s commitment protocol model [29].
3.1 Communication Artifact

We follow the ontological model for organizational roles proposed in [7, 8], which is characterized by three aspects: (1) **Foundation**: a role must always be associated with an institution it belongs to and with its player; (2) **Definitional dependence**: the definition of the role must be given inside the definition of the institution it belongs to; (3) **Institutional empowerment**: the actions defined for the role in the definition of the institution have access to the state of the institution and of the other roles, thus, they are called *powers*; instead, the actions that a player must offer for playing a role are called *requirements*.

*Communication artifacts* realize a kind of mediated interaction that is guided by commitment-based protocols. Figure 2 shows the UML schema of the supertype of communication artifacts implementing specific interaction protocols (e.g., Contract Net, Net Bill, Brokering): the *BasicCommitmentCommunicationArtifact*. We call an instance of an artifact of type BasicCommitmentCommunicationArtifact an *interaction session*. It represents an on-going protocol interaction, with a specific social state that is observable by the interacting agents, that play the protocol roles. The BasicCommitmentCommunicationArtifact presents an observable property, *Roles*, that is the collection of the roles of the protocol (definitional dependence [7, 8]). Actions have a social effect only when they are executed by the role they are assigned to, but actions are not defined at this super level, rather they are provided by the instantiations of the BasicCommitmentCommunicationArtifact, i.e. by artifacts implementing specific protocols. Each protocol action is implemented as a public *operation*, which is associated...
Fig. 2. The UML Class diagram for the commitment-based communication artifact.

to a role by means of an operation guard (institutional empowerment [7, 8]): the
 guard checks who is performing the operation; if the agent is not the one play-
ing the right role, the action simply has no effect, otherwise, the fact that the
 action was executed is registered in the social state together with its meaning.
 An action can have some additional guards, implementing context preconditions:
 this condition specifies the context in which it makes sense that the action pro-
duces the described social effect. An artifact can be monitored by an observer
 agent, that, following the CArtAgO terminology, is focusing on that artifact, particu-
larly on one or more public properties. A change of one of these properties causes a
 signal, from the artifact to the observer agents, about the property
 that changed: the agents perceive the new artifact state. In particular, when the
 creation of a commitment, involving an agent as a debtor, is signaled to it, this
 agent is expected to behave so as to satisfy the commitment. The agent is free to
decide how (and if) it will handle the satisfaction of its commitments. Therefore,
the requirement is that an agent has the capability to behave so as to achieve
the involved conditions [7, 8]. An agent who does not show such capabilities is bound to violate its commitments.

BasicCommitmentCommunicationArtifacts provide a property, tracking the identity of the agents actually playing the various role. Two operations are provided in order to manage the association between an agent’s identity and a role: enact(Role role) and deact(Role role), by means of which an agent can explicitly assume/cease a protocol role (foundation [7, 8]). After enacting a role, the use of the associated operations on the artifact will have social consequences.

The communication artifact has an observable property, social state, that is a set of zero or more elements of type Commitment or Social Fact. As we can see in Fig. 2, these structures are simple Java objects, representing the actual social state. The artifact is responsible to manage the Social State structure, i.e. the Commitments life-cycle, as well as the assertion or retraction of social facts, via methods called on commitment and on social fact objects. For Commitment management, we refer to the basic operations of commitment manipulation [29]: create, discharge, cancel, release, assign, delegate. The operations regarding the commitments life-cycle are implemented as artifact internal operations, therefore, the agents cannot modify them explicitly. The communication artifact exposes the social state, whose evolution is controlled by the agents via the protocol-provided actions. Finally, communication artifacts provide service operations, which can be performed only by the ArtifactManager Agent (see below) for managing the protocol roles and the identities of their players.

When the social state property changes, due to the execution of a protocol action (an artifact operation) on the communication artifact, all of the agents using the artifact will be notified, allowing them to react (or not) to the evolution of the interaction. This mechanism is a core part of the CArtAgO framework.

The ArtifactManager Agent plays the role of a Yellow Pages Agent for communication artifacts, or, in other terms, of an artifact broker. It has a crucial role: it is a “communication channel” broker, gathering requests for both focused or broadcasting calls for interaction. As such, it provides a collection of utility services. It supplies information about the interaction protocols (e.g. it provides the XML describing a given protocol, it allows a search for a protocol, a list of active communication channels, a list of interacting agents); it answers to requests about the status of an existing interaction session; it notifies the subscriber agents a particular session availability, and so on. Its main purpose is to prepare the communication artifact among the interacting agents, and to supply it to the requesting agents. It can also enable other interested agents to monitor, audit, or, more generally, observe the social state evolution. The communications between the ArtifactManager Agent and the requesting agents is realized via FIPA-ACL messages: when a requester sends a request ACL message to the ArtifactManager Agent, specifying the protocol and the role it wants to enact, the latter will do the following steps:

1. Check if the requested protocol is available;
2. Check if the requested role is foreseen by the protocol;
3. Create/retrieve a communication artifact of the requested type;
4. Set the requested artifact role field to the agent identifier (AID) of the requester;
5. Respond to the requester with the artifact’s reference;
6. Possibly inform other interested agents of the availability of the communication artifact.

The initialization procedure is modeled as a simple FIPA Request Interaction Protocol, where the content of messages consists of the communication artifact request parameters. After this phase, the agent can use the exact operation to start playing the requested role. The use of an agent does not necessarily imply a centralization of the yellow pages: agents may directly create communication artifacts; yellow pages can be federated.

### 3.2 Using Mediated Communication at Runtime

In the following, we show a scenario in which a communication artifact is used, to better explain how to leverage the communication artifacts and the ArtifactManager Agent. We adopt the well-known FIPA Contract Net Protocol (CNP), modeling it as a commitment-based protocol and implementing a corresponding artifact. The scenario is depicted in Fig. 3.

![Fig. 3. The interaction between the main elements of our proposal, in a CNP example.](image)

The JADE infrastructure is extended with the ArtifactManager Agent, that provides a Yellow Pages service for communication artifacts. It can respond to
ACL Messages, that encode requests of a new Communication Artifact, either with a *Failure* message or an *Agree* message. In the latter case, it will either prepare a new instance of the requested communication artifact, or it will return an already existing artifact. For instance, suppose that agent $A1$ has to assign a task, and agents $A2$ and $A3$ have the capability of performing it. Suppose that $A2$ and $A3$ already registered to the ArtifactManager Agent ($ArA$ for brevity), and that this has already instantiated a Contract Net Protocol communication artifact ($CNPCA$ for brevity). At this time, the (partial) state of $CNPCA$ is:

- **Initiator**: null
- **Participants**: \{A2.$AID$, A3.$AID$\}

where $AID$ is the JADE Agent Identifier. $A1$, then, asks $ArA$ for a $CNPCA$, following the procedure described before, without specifying a particular participant. $ArA$ matches this request with the already prepared $CNPCA$: the match is successful, inasmuch the Initiator role is not played by any agent. So, $ArA$ stores $A1$.AID in the *Initiator* property of $CNPCA$, and returns its reference to $A1$. Following the CArtAgO terminology, agents $A1$, $A2$ and $A3$ focus on the *SocialState* property of $CNPCA$ immediately after having its reference. This means that any change to the social state will be signaled to the three agents, who can take decisions accordingly. The agents interact with one another via operations on $CNPCA$, and observe the social state evolution in order to reason about which actions to take.

An agent can stop playing a protocol role at anytime by executing the *deact* operation. The artifact unregisters its AID from the AID-role mapping list. On the other hand, an agent may enact a partially executed role within an interaction session. What about commitments in such cases? In this work we focused only on the communicational and interaction-related aspects of playing protocol roles: sanctions or other action concerning the institutional (or organizational) levels are not accounted for yet. Simply, since responsibilities are associated to roles, deacting a role yields that the resigning agent will not need anymore to fulfill them, while a substituting agent needs to accept the current commitments of the role it is assuming [29]. A reference model to include, in the future, also institutional aspects could be the JaCaMo proposal [9].

### 3.3 Using Mediated Communication at Design Time

We assume that MAS designers know a collection of communication artifacts, each representing a commitment-based protocol. Each protocol is enriched with an XML-based description of it, a *Protocol Manual*, available both at design- and at run-time. It is an add-on to the CArtAgO artifact manual, with orthogonal scopes and purposes. It can be used by MAS and agent designers as a guideline for understanding whether an agent is suitable for a protocol role as well as for understanding whether a protocol role suits the purposes of an agent. From a methodological point of view, the designer needs the Protocol Manual to know the social consequences of the actions supplied by an artifact, in terms of social...
facts and commitments, so he/she can design agent behaviors accordingly. Then, depending on the implemented behavior, the agent will decide how to use information about the social state evolution, how to fulfill commitments, which social action (i.e. a public artifact operation) to execute and when. Ideally, the designer should equip the agent with the behaviors that are necessary to bring about the conditions of the commitments it will possibly take. This protocol-centric design, jointly with the commitment nature of protocols, avoids a critical facet of JADE protocols. Here, a pattern of interaction is projected on a set of JADE behaviors, one for each role, thus making a global view of the protocol and its maintenance difficult, and binding the very interaction to ad-hoc behaviors. Consequently, the risk of conflicting behaviors, not devised at design time, increases. This way, the designer can leverage a library of programmable communication artifacts, focusing on the internal agent behavior without being concerned about ad-hoc shaped communication behaviors.

4 JADE Methodology revised

We use the JADE Methodology [22] to model a real-scenario MAS, we call FinancialMAS. For brevity, we show only the fundamental steps needed to draft the system and to highlight the benefits of reifying commitment-based protocols by means of artifacts, and thus based on mediated interaction.

The JADE Methodology is a JADE founded agent-oriented software engineering methodology. It proposes a fully agent-based approach, instead of adapting Object-Oriented techniques (like MASE [28], Adelfe [6] or MESSAGE [10]). It concerns the analysis and the design phases of the software development life cycle. The methodology considers agents as “pieces of autonomous code, able to communicate with each other” [22], thus following a weak notion of agency; it does not account for mentalistic/humanistic agents properties.

In the analysis phase, the first step is the identification of use cases, i.e. functional requirements of the overall system, which are captured as standard Use-cases UML Diagrams. Starting from this, the designer can point out an initial set of agent types: an agent type for each user/device and for each resource. The agent paradigm foresees that even external devices and software/hardware resources (e.g. legacy systems, databases, external data sources) are represented with an agent. The designer, then, identifies responsibilities, i.e. the activities provided by system each agent is responsible for; and acquaintances, that is relationships between agents aimed at fulfilling some responsibility. The results are a Responsibility table and an Agent diagram with initial acquaintances. No distinction is made between acquaintances and responsibilities: in fact, the mentioned table will contain both. The analysis is completed by executing activities related to agents/acquaintances refinement, to define discover services and to add management/deployment information. The design phase starts with the interaction specification step, where an interaction table is produced. It refers to the responsibility table in order to define interactions between JADE agents, specifying the interacting agents, the protocol and protocol role (e.g. Initiator

26
or Responder), the reference responsibility, and a triggering condition. It is suggested to use, when possible, standard JADE protocol behaviors, that must be added to an agent’s behavior set to implement the corresponding protocol role. The subsequent steps focus on the specification of agent interactions with users and resources; the definition of a yellow page services, using the JADE Directory Facilitator; the implementation of agent behaviors, starting from JADE protocol behaviors related to responsibilities. A last effort is the definition of a shared, system-wide ontology.

Table 1. Responsibility Table for FinancialMAS.

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>No.</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investor agent (IA)</td>
<td>1</td>
<td>Let investor search for investments proposals</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Assist investor in setting search parameters and data</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Support the individuation of the investor’s risk profile</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Support in proposal acceptance</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Withdraw from an investment contract</td>
</tr>
<tr>
<td>Financial Promoter agent (FP)</td>
<td>1</td>
<td>Respond to investment searches</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Assist financial promoter in risk-classifying financial products</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Determine the investor’s profile</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Support individuation of the investor’s risk profile</td>
</tr>
<tr>
<td>Bank agent (BA)</td>
<td>1</td>
<td>Support bank in investment contract subscription</td>
</tr>
<tr>
<td>Financial Provider agent (FV)</td>
<td>1</td>
<td>Support and provide financial information</td>
</tr>
<tr>
<td>Integration agent (IntA)</td>
<td>1</td>
<td>Serve and support integration with legacy bank informative systems</td>
</tr>
</tbody>
</table>

By applying the steps of the methodology, we obtained an initial design prototype for FinancialMAS, concerning an initial set of agents and the so called responsibility table (Table 1). In the terminology of the JADE Methodology, responsibilities amount to functional duties, agents are responsible for, from an overall MAS point of view. To handle them, agents possibly need to interact with one another. The result of this kind of analysis is synthesized in an Interaction table (Table 2). At this point, instead of realizing protocols via distributed JADE behaviors, we implement them via commitment-based communication artifacts. We assume to have already designed artifacts for common interaction patterns (like Contract Net Protocol, Query Protocol and Request Protocol), thus shaping the MAS interaction patterns via reified commitment protocols. The resulting model is depicted in Fig. 4, whilst in Fig. 5 we zoomed into the implementation of one of the commitment artifacts, the Contract Net Protocol artifact.

In Fig. 6 we highlight the very same protocol, implemented via pure JADE behaviors. Looking at the picture, the reader can perceive a major drawback of the latter approach: being part of an interaction protocol entails the adoption of an entire behavior, that must be added to the set of the internal agent behaviors. The resulting agent design breaks the autonomy of the agent, since the agent has an additional behavior for each role of each interaction it takes part to, increasing the possibility of conflicts between behaviors, and increasing the overall agent design complexity. Furthermore, this approach hinders the observability of the
interaction, unless the designer adds specific sniffing or audit agents to log every message passed. In performance-critical applications, having more agents and producing a message overhead can produce undesirable scenarios.

Table 2. Interaction Table for FinancialMAS: who interacts with whom, to fulfill which duty, by using which protocol.

<table>
<thead>
<tr>
<th>Interaction Type</th>
<th>Protocol</th>
<th>Role</th>
<th>With</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investor Agent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Investment</td>
<td>CNP</td>
<td>Initiator</td>
<td>FP</td>
<td>Investor searches an investment</td>
</tr>
<tr>
<td>Profiling</td>
<td>Query</td>
<td>Participant</td>
<td>FP</td>
<td>Investor chose a Financial Promoter</td>
</tr>
<tr>
<td>Proposal Acceptance</td>
<td>Query</td>
<td>Participant</td>
<td>BA</td>
<td>Investor chose a financial product</td>
</tr>
<tr>
<td>Withdraw</td>
<td>Request</td>
<td>Initiator</td>
<td>BA</td>
<td>After Investor accepted a proposal</td>
</tr>
<tr>
<td>Financial Promoter Agent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Respond to Search</td>
<td>CNP</td>
<td>Participant</td>
<td>IA</td>
<td>Investor searches an investment</td>
</tr>
<tr>
<td>Profiling</td>
<td>Query</td>
<td>Initiator</td>
<td>IA</td>
<td>Investor chose a Financial Promoter</td>
</tr>
<tr>
<td>Bank Agent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proposal Acceptance</td>
<td>Query</td>
<td>Initiator</td>
<td>IA</td>
<td>Investor chose a financial product</td>
</tr>
<tr>
<td>Withdraw</td>
<td>Request</td>
<td>Participant</td>
<td>IA</td>
<td>After Investor accepted a proposal</td>
</tr>
<tr>
<td>Financial Provider Agent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instead of basing interaction design directly on JADE behaviors, we propose a clear notion of Role that an agent must enact to participate in an interaction session, so the designer must only implement the behaviors for fulfilling the commitments caused by the execution of a protocol action. Playing a role gives an agent powers, in terms of social state modification (i.e. the state of the interaction session) as a consequence of its actions, and the agent designer can use them if, when and how he/she wants. This approach is illustrated in Fig. 5, where the commitment-based definition of CNP is provided. The protocol consists of a set of social actions, each of which has an impact on the social state of the interaction. For example, when an agent playing the role p (participant) executes the artifact action propose, the social state is modified by creating the commitment \( C(p, i, \text{accept, done} \lor \text{failure}) \). This change is signaled to the agent playing the role i (initiator), who will handle it in some manner (depending on its behaviors) and decide whether accepting the proposal of that participant. Instead, when a failure is executed the raised event automatically discharges a commitment created by a propose.

5 Related works, discussion and future work

2COMM is a first step towards the implementation of the ideas proposed in [3], realizing a programmable communication channel by means of artifacts, which is interaction-centric, exploits the social meaning of interaction supplied by commitment protocols, and enables monitoring functionalities. The use of commit-
ments gives a normative value to the encoded protocol, while the act of using a communication artifact amounts to the explicit acceptance, by the agent, of the rules of the protocol. The proposal conjugates the flexibility and the openness that are typical of MAS with the need of modularity and compositionality that are typical of design and development methodologies. The realization of commitment protocols as artifacts/environments is an advancement of research on
commitment-based approaches, w.r.t. approaches like [13], where these elements reside in the middleware, and are shielded from the agents and from the designer.

We believe that a commitment approach brings relevant advantages in terms of design and modeling flexibility, modularity and traceability. The resulting artifact explicitly provides a notion of Role that is decoupled from the interacting agent, instead of cabling it into an agent behavior (as in the JADE Methodology) or of composing different atomic roles to build an agent type (as in the GAIA Methodology [31]). Both approaches break into inner agent definitions, hindering the agent autonomy and the openness of the system. The artifact entity supplies a natural way for logging and audit purposes, leveraging the concept of social state (and its evolution). In a pure agent environment (like JADE), a similar result is obtained via a massive use of either message-sniffing agents and/or auditing agents, with a consequent overhead of the number of messages that are passed. This is, for example, the case of the proposal in [21]. By being an observable property, the social state provides the agent society a clear vision of who is responsible of what, in which protocol interaction, and when an agent acted so as to fulfill its commitments.

2COMM focusses on the interaction protocol layer, leaving aside issues concerning the society of agents in which the interaction takes place. Thus, it does not, for instance, tackle how to deal with violations of commitments. In order to properly handle these aspects it would be interesting to combine its use with proposals from the area of e-institutions. Concerning this field 2COMM would provide an improvement in that it would introduce the possibility to account for indirect forms of communication. As [18] witness, there is an emerging need of defining a more abstract notion of action, which is not limited to direct speech acts, whose use is not always natural. For what concerns organizations, instead,
there are some attempts to integrate them with artifacts, e.g. ORA4MAS [19] and JaCaMo [19] http://jacamo.sourceforge.net, which also accounts for BDI agents. Following the A&A perspective, artifacts are concrete bricks used to structure the agents’ world: part of which is the organizational infrastructure, part amounts to artifacts introduced by specific MAS applications, including entities/services belonging to the external environment. In [19] the organizational infrastructure is based on Moise\(^+\), which allows both for the enforcement and the regimentation of the rules of the organization. This is done by defining a set of conditions to be achieved and the roles that are permitted or obliged to perform them. The limit of this approach is that it cannot capture contexts in which regulations are, more generally, norms because norms cannot be restricted to achievement goals.

Finally, we think that our proposal can give significant contributions to industrial applicative contexts, in particular for the realization of Business Processes and in particular of human-oriented workflows, whose nature is intrinsically social and where the notion of commitment plays a fundamental role [20].

References

Applying an O-MaSE Compliant Process to Develop a Holonic Multiagent System for the Evaluation of Intelligent Power Distribution Systems

Denise Case and Scott DeLoach

Department of Computing and Information Sciences
Kansas State University,
234 Nichols Hall, Manhattan, Kansas, USA.
{dmcase,sdeloach}@ksu.edu

Abstract. This paper describes the application of an Organization-based Multi-agent System Engineering (O-MaSE) compliant process to the development of a holonic multiagent system (MAS) for testing control algorithms for an intelligent power distribution system. The paper describes the Adaptive O-MaSE (AO-MaSE) process, which provides architects and developers a structured approach for testing and iteratively adding functionality in complex, adaptive systems. The paper describes the holonic MAS architecture for the intelligent power distribution system, the challenges encountered while developing the holonic architecture, the lessons learned during the project, and demonstrates how the application of the process enhanced project development.

Keywords: Agent-oriented software engineering, holonic multi-agent systems, adaptive systems, smart infrastructure, intelligent power distribution systems

1 Introduction

Power distribution systems (PDS) are estimated to account for approximately 40% of the capital investment in power systems worldwide, roughly comparable to the amount invested in generation, and about twice that invested in transmission assets [19]. PDS automation has lagged the advances in generation, due in part to the distributed nature and the massive number of components that make up the system. Power distribution begins with the primary circuit leaving a substation. It includes a distribution network of 3-phase feeder lines that branch into single-phase lateral lines and a variety of supporting equipment. Lateral lines distribute power through shared transformers that ultimately feed a set of electricity consumers, such as individual homes. Traditionally, control of the system, like the energy, has flowed from the central power source outward and downward toward the end consumer.

However, the increasing presence of renewable, distributed energy resources creates a bidirectional flow of power in the system. Rather than solely consuming power, home owners are installing increasing numbers of rooftop photovoltaic (PV) systems.
that generate power from solar radiation. The PV electricity generated during peak hours of the day can be greater than what is consumed by the associated home and creates an opportunity for homeowners to sell excess power into the grid. Distributed energy suppliers introduce a need for enhanced information flows, including online auctioning of power between growing numbers of market participants. At the same time, distributed energy sources are subject to intermittency. Wind fluctuation and passing clouds can introduce rapid variation in the amount of power flowing into the system. Rapid change in generation creates significant challenges for maintaining voltages within desired ranges. However, coordinated volt/var control can help provide consistent voltage to consumers while reducing rapid cycling of equipment, improving efficiency, and reducing the overall cost of power generation.

Centralized control is supported by load tap changers (LTC) near the substation that have a limited number of setting changes per day (more rapid cycling is expensive and shortens the equipment life). Distributed control can be supported by the addition of smart inverters that moderate a PV system’s reactive power to offset generation changes and by line capacitors employed in various places along the grid. Distributed control equipment coupled with centralized, large impact devices like the LTCs create an excellent opportunity for distributed intelligent systems that can adapt reactively and proactively to offer substantial benefits. In addition, distributed intelligent systems can help the increasingly connected remote nodes that are both electricity producers and consumers (prosumers) work together when disconnected from the grid. During this islanded mode participants can work together to provide electricity to critical loads, even though the lack of a stable power source (i.e., power from the grid) creates additional challenges for power supply and quality.

Multiagent systems (MAS) are getting significant attention for PDS. In addition, there is a growing interest in holonic multiagent systems (HMAS) for PDS [19]. HMAS are adaptive, communicative, and autonomous – traits they receive from their MAS heritage – and their hierarchical, recursive structure is a natural fit for PDS systems. Holonic comes from the Greek word holon, a union formed from holos meaning whole and on meaning parts [19]. Thus, holons are parts that are also wholes. In an HMAS, the holon indicates an agent participating in one organization that also represents an entire organization itself. The overall organization of nested holonic organization agents is called a holarchy [8]. This hierarchical composition mechanism defines a powerful framework for distributing intelligence and finding local solutions in a recursive manner.

Thus, the overall goal of the Intelligent Power Distribution System (IPDS) project at Kansas State University is to demonstrate an HMAS architecture that is capable of adaptively controlling future PDS that are expected to include a large number of renewable power generators, energy storage devices, and advanced metering and control devices. Specifically, we are considering PDS with a high penetration (between 25% and 75%) of home-based PV systems. The HMAS architecture will also be used to support new analytical algorithms aimed at limiting the impact of information delay, quality and flow on the PDS. The purpose of this paper is to present the O-MaSE compliant software engineering process that we used in developing our prototype HMAS architecture for the IPDS project.
A mapping from our prototype HMAS architecture to a physical distribution system is shown in Fig. 1. The left side of Fig. 1 illustrates the HMAS holarchy of adaptive agent organizations and the right side illustrates the physical system. The lowest level of the holarchy shows peer agents representing single homes acting within a local organization at the neighborhood level (assumed to be represented by the pole transformer serving that set of homes). Each neighborhood organization is represented by a corresponding neighborhood organization agent in the higher lateral-level organization. Nested organizations reflect the physical system from the originating substation down to individual consumer homes. The holarchy, like the physical network, recursively aggregates the systems and organizations to provide an integrated model of the system and provides a framework that allows the IPDS be able to learn and support adaptive, distributed control.

2 Related Work

2.1 Smart Infrastructure Optimization with Agents

Smart infrastructure optimization involves some of the most complex and critical systems in modern society [19]. Agent technology offers a way to manage the inherent complexity of such systems. Agents can be used to represent simple variables in a computer program as well as complex, distributed, intelligent objects involving potentially infinite numbers of states, decisions, and actions and reactions [23]. When modeling power systems, we are especially interested in agent traits such as autonomy, heterogeneity, adaptivity, social ability, communicability, flexibility, and concurrence.
Agents implement goal-based behavior and IPDS agents must demonstrate the ability to support the objectives of their respective owners while also acting cooperatively to achieve common objectives, such as maintaining critical loads and system efficiency.

Current research projects include a variety of studies involving the application of MAS to power systems, with active research projects focusing on power auctioning, negotiating, volt/var control, distributed communications, and other focus areas [3, 11, 15, 17, 20, 21, 22]. Recent research has also applied HMAS to PDS [2, 12, 14], sometimes in concert with specific agent-oriented software engineering methodologies. Power flow, quality, and control lends itself to distributed, recursive optimization where possible. Some local optimization can be distributed and may not result in propagation throughout the hierarchy, while the system as a whole may be impacted by larger, more centralized control options such as load tap changes. Using a flexible, holonic architecture will allow us to evaluate a variety of control algorithms and strategies.

2.2 Agent-Oriented Software Engineering Methodologies

Several agent-oriented engineering methodologies have been defined for developing complex, adaptive systems. These methodologies include metamodels and process flows that provide the structure necessary to engineer MAS. Some of the methodologies that have been successfully employed include ADELFE, ASPECS, INGENIAS, O-MaSE, PASSI, Prometheus, SODA, and Tropos [7]. Some methodologies, such as ASPECS and ANEMONA support holonic concepts [1, 6, 10] while O-MaSE supports hierarchical decomposition via organizational agents. However, O-MaSE was chosen for this project due to its flexibility and tool support, which includes the agentTool3 integrated development environment.

3 Background

3.1 O-MaSE Process Framework

O-MaSE is an organization-based, role-centered process framework that consists of three main components: a metamodel, method fragments, and guidelines [9]. The metamodel describes system components as shown in Fig. 2. Method fragments define engineering roles, their activities, and the resulting work products, such as the goal models, role models, and plan diagrams that define the system. Each aspect of O-MaSE is supported by agentTool3 modeling tools, which support method creation and maintenance, model creation and verification, and code generation and maintenance.

Linnenberg et al. used the O-MaSE methodology and agentTool3 to develop DEMAPOS (DEcentralized MArket Based POwer Control System) [16] for power trading and we have followed the DEMAPOS convention of combining entities capable of producing and/or consuming electricity into the notion of prosumer agents.
3.2 Holonic Multiagent Architectures

Holonic multiagent architectures introduce additional elements to MAS. Cossentino et al. describe the multilevel interplay between local organizations [6] where an agent participating locally in an organization may represent the entire lower-level organization while also participating as an agent in a higher-level organization. A holonic organization agent may play different roles in different organizations simultaneously as shown in Fig. 3. Participation in multiple organizations is not unique to holonic agents; rather it is the recursive representation of progressively distributed agent organizations that made the holonic approach desirable for evaluating IPDS.

We evaluated a variety of existing systems to provide core MAS functionality and selected the Organization-based Agent Architecture (OBAA) as our foundation.

---

Fig. 2. The O-MaSE Metamodel (multiplicities not shown).

Fig. 3. A holonic organization agent (OAgent) serves as the master agent in its lower-level organization (blue) and a peer agent in a higher-level organization (purple).
OBAA has been employed in a variety of adaptive systems [18]. OBAA agents include a shared control component enabling basic communication capabilities within the local organization coupled with an embedded knowledge representation of the organization in which the agent operates. In OBAA, each control component coordinates with the team and maintains a complete copy of the local organization knowledge. Each OBAA agent also has a domain-specific execution component that manages roles and capabilities as shown in Fig. 4.

The primary objective of the generic OBAA architecture is to offer a general framework of classes from which domain-specific adaptive systems can be built. The OBAA framework is built on the Organization Model for Complex, Adaptive Systems, OMACS [9], and includes an executable goal model, GMODS [9]. The control component parts provide the functionality for agents to get initialized, register with an acting supervisor and, once the registration process is complete, to begin executing their assigned tasks. An initial agent registration process was defined to get the system running; it will be enhanced to incorporate leader election as the project continues.

4 AO-MaSE: An O-MaSE Compliant Process

O-MaSE provides a foundation supporting tailored software engineering implementations. A compliant process must meet the following requirements: (1) no new constraints may be placed on existing entities and relationships in the O-MaSE metamodel, (2) the method guideline pre-conditions must not become stronger or post-conditions made weaker, and (3) no existing metamodel entities, tasks, work products, or method-roles may be eliminated [9].

MAS and HMAS are complex models of complex systems. Getting started with such frameworks can be challenging [13]. The Adaptive O-MaSE software engineering process (AO-MaSE) provides a set of recommendations for dealing with that complexity by applying some of the principles commonly associated with agile pro-
cesses [4]. Several of these agile principles are associated with MAS in general and include an ability to respond to changes, an ability to participate in ongoing collaboration, a recognition of the importance of interaction between autonomous participants, and a focus on goal-driven, executable components. In a similar way, the AO-MaSE approach focuses on adaptability and the structured evolution of a working system. Other systems such as PASSI have gone further to incorporate agile processes [5].

In AO-MaSE, the architect begins by creating a fully executable but limited scope vertical spike through the system to create a working version early which offers a solid core from which increasingly complex analyses and behavior can evolve. AO-MaSE follows the O-MaSE compliant process includes three iterations that includes the tasks and work products shown in Fig. 5. The process follows four key strategies:

- Start simply and add incrementally; in the OBAA framework, agents given even simple goals require a significant infrastructure to execute.
Apply recommended process conventions to enhance clarity and consistency.
Follow models with code construction to get working systems early.
Expand, enhance, and refactor as functionality evolves.

By following the AO-MaSE approach and detailed implementation guidelines, the full set of required components can be implemented early, and form a basis for expanding and enhancing the system. Completing the connections in a working system provides examples of how components connect the various models and drive the behavior of the system. For example, event triggers on the goal model may appear as transitions in the plan diagrams and domain objects may appear in method parameters in plan states and associated capabilities. A working version that connects the parts provides a concrete example for software engineers and developers that have little experience in agent-oriented engineering. The method construction guidelines provide the ability for a team of software engineers and subject matter experts to work collaboratively to select key elements for implementation and to develop associated work products including requirements specifications, goal models, organization models, domain models, role models, role plans, plan states, capabilities, protocols, policies, and code.

5 Applying AO-MaSE in the Development of IPDS

The AO-MaSE process was developed and implemented during the production of the initial IPDS architecture. AO-MaSE design conventions, recommended practices, and guidelines are described and illustrated.

5.1 Iteration 1 – Getting Started

In addition to the infrastructure of the component parts, an OBAA-based system offers significant initial agent functionality, but introduces some additional embedded complexity. System behavior develops in response to a variety of events such as goal triggers, agent registrations, and organizational events. Early execution can help software engineers get a better understanding of the system.

The first iteration results in a streamlined implementation that provides an early executable model of the system. The following summarizes the AO-MaSE recommended practices for an initial iteration.

1. Define one top-level goal to reflect the overall behavior desired by the system; add a small number of terminal goals (without subgoals) to represent core objectives.
2. Define the initial set of interfaces to the overall organization.
3. Define roles to achieve each terminal goal.
4. Define plans to perform each role.
5. Define capabilities specific to each plan; define a local domain-specific communication capability and an external controller communication capability.
6. Assign role requirements. Most roles require control communication (for OBAA agents), the local domain-specific communication, and the appropriate
role-specific capability. Certain roles will require external controller communication capability.

7. Define a limited number of plan states, e.g. *INIT*, *role-specific*, and *STOP*.
8. Define plan state transitions and state behaviors by defining and calling capability methods.
9. Define agent classes based on the problem domain.
10. Configure agent instances with associated capabilities and attributes.
11. Configure environment object instances such as sensors and actuators.
12. Implement code components by extending the OBAA framework.
13. Configure, implement, debug, test, and execute the initial vertical spike.
14. Where possible, follow parallel, explicit naming conventions that differ only in type (as shown in the IPDS models).

The project began with the specification of requirements. From the set of requirements for the initial phase of the project we selected an initial focus that allowed us to test core functionality – the distribution and management of goals within a local organization.

Since achieving each goal requires substantial infrastructure, we started with a small set of core objectives. The top goal of each recursive organization is Support IPDS, as shown in It will guide agent organizations while connected to the grid and while running in islanded mode.

The organization model was developed to define the boundaries and interfaces for the system. Each IPDS would seek an external controller that would both receive requests and send requests/guidelines down to the system, enabling centralized control and communications from the primary energy supplier. Inputs were provided to characterize the organization’s goals. The goal model was drafted and then refined to show the supervisor triggering a manage instance goal for each participant. The domain model began to reflect the objects in the environment and included a smart meter object and a PV system, along with equipment attributes and unique identifiers.

Following the guidelines, we created a role for each terminal goal, a plan for each role, and gave each plan three initial states: (1) *INIT* for performing actions that will only need to be done once, (2) a role-specific state that captures the main work of the role, and (3) a *STOP* state consisting of behaviors to be executed when finishing the plan. The recommended capabilities were defined. As plan states were developed in the plan diagrams, we were specifying the methods required of each capability. Parallel naming conventions for goals, roles, plans, and role-specific default capabilities aided clarity and were used to employ additional code automation. Agent classes did not parallel the goal or plan names. Instead, they reflected the physical installation or focus of the agent type. We began with a Neighborhood Agent class, expected to run on or near a transformer serving 2-6 homes, and a Prosumer agent class, expected to be installed on or near a home-based smart meter.

As the OMACS components developed, they were implemented in the OBAA-based IPDS framework. Agent and Environment configuration files were used to instantiate specific agents and objects for a variety of test cases.
The OBAA framework can be employed immediately if one control component master is declared for any local organization. We began with one supervisor neighborhood agent (the control component master) and two prosumer agents (both control component slaves) to test the ability of the system to solve adapt to changing local conditions. Some key models from Iteration 1 are shown in Fig. 6. The models illustrate the parallel naming conventions between goals, roles, and plans, and although only a small subset of the models created are included, helps illustrate the infrastructure support underlying an agent-based adaptive system.

5.2 Iteration 2 – Filling in the Framework

With a working simulation provided during Iteration 1, the focus in Iteration 2 shifted to adding functionality to address a variety of potential challenges. We began working with the new holonic organization agents and the development focused on enhancing the plan states and capability method calls. Additional capability classes were added, providing additional differentiation and room for expanded functionality. Capabilities
were implemented with simple algorithms that served to define the expected interfaces that would be required to support more complex optimization algorithms that were being developed in parallel research projects.

The goal model was enhanced to include parameterized goals with the external controller providing combined guidelines for the organization. Additional triggers were added to the refined goal model. The supervise goal, which had been distributing combined goals among participants during the INIT state, was enhanced to adapt participant goals during the SUPERVISE state in response to each participant’s simulated history.

Organization guidelines were grouped into objects with defined purposes, making the system easier to expand as requirements were added. Three types of guidelines were given to each organization: combined load guidelines, combined power quality guidelines, and evaluation guidelines that reflected desired feedback intervals and forecast horizons. As a holarchy, the combined organization guidelines could be adapted in response to temporal conditions just as local participant guidelines were adapted. Plan states continued to evolve to reflect more complex logic and additional actions and events were added to define the transitions between states. Objects and attributes were added to the domain model as more external devices were defined.

Capabilities grew in functionality as plan state logic developed. Capability methods were enhanced to include simulation interfaces and smart meter sensor capabilities began obtaining simulated device data from MatLab®. As capabilities became more complex, they were refactored into smaller, more specific capabilities that in turn began to grow in functionality. An IPDS Builder component was added to support the reliable generation of test cases.

### 5.3 Iteration 3 – Extending Functionality

The third iteration focused on extending the refined goal model; introducing forecasting goals and adding supporting agent types. Although goal changes represent a relatively major change to the IPDS design, by following the guidelines and recommended process and code policies, we were able to add new features more easily. Additional goals brought additional triggering events and goal parameters. The expanding goal models and role models are shown in Fig. 7.

As communications are added to plan diagrams, they include the specification of the performative, the type of message content, and the role of the agent with which the communication takes place. Message classes and their associated message content classes were implemented for each communication capability.

As an IPDS organization starts up, agents participating in the organization register with the control component master. The specification goal tree gets instantiated and activates the top level goal along with any non-triggered, non-preceded leaf goals. For example, as the goal plan for the Supervise Prosumers goal is executed, the Supervise Prosumers Plan INIT state triggers an instance of the Manage Prosumer goal for each participant. As each home agent gets assigned to a Manage Prosumer Role, it first enters the Manage Prosumer Plan INIT state, and then triggers a new instance of an associated Forecast Prosumer goal.
The home agent then transitions to the Manage Prosumer MANAGE state and begins sensing consumption and generation readings, which it reports back to the Supervisor, alerting the Supervisor if it detects an out-of-bounds condition. The supervisor optimizes combined local guidelines within the organization, adapting participant goals to maximize compliance. If guidelines cannot be met within the local organization, the supervisor will raise a request to the external controller who will, in turn, attempt to address the request from within the controller agent’s local organization, recursively raising requests up the holarchy until a solution is available.

Fig. 8 shows a view into a running IPDS system. There is one debug window for each organization agent. The organization goals appear in each top left panel. Roles appear in the top center panel. Participating agents are shown in the lower right panel. In the center bottom panel, the assignments are displayed, indicating that each agent has been assigned to a specific instance goal based on their capabilities and attributes as defined in the agent configuration file. In this assignment panel, we can see the current values of the agent’s goal parameters as they are adjusted by their local organization supervisor. At this point in the simulation, the prosumer goals are being distributed in accordance with each participant’s demand. Maximum kW guidelines may
be positive or negative. Negative upper boundaries can be assigned to a participant generating more PV power than the participant is consuming.

The extended system has passed a variety of tests and demonstrates the operation of multiple goal models and assignments. During this most recent iteration, the system has been extended to additional levels of the holarchy in preparation for the evaluation of the initial IEEE test case, involving nearly 40 local IPDS organizations and approximately 100 homes.

6 Software Engineering Challenges

Architecting complex, adaptive systems can be challenging. Employing existing components can facilitate design, but a true understanding of the interrelationships between framework components may take a while to develop. Designing a new organization-based HMAS for IPDS required developing a detailed understanding of the O-MaSE metamodel, the OBAA framework, and GMoDS, and understanding the connections between the various elements, how they were related to code structures and most importantly, how they drive behavior during execution. Initial attempts took longer than expected. The AO-MaSE process was developed as a way to illustrate the system design in a concrete manner and allow new team members to become productive faster. The guidelines have proven helpful. Additional functionality can be added more quickly and software engineers can read and implement model designs more efficiently. Standards have allowed greater automation and agentTool3 has been updated. Debugging configuration files resulted in the creation of IPDS builder factories.

Fig. 8. Simple IPDS Holarchy (Substation, Feeder, Laterals, Neighborhoods)
that will be crucial as we grow test cases. The guidelines recommended have been beneficial in helping newer project members get involved and productive more quickly. They support more rapid response to new requests by the interdisciplinary project team. Strict naming and documentation requirements improve clarity and consistency. Although the number of classes required is substantial, the focus of each is such that enhancements can proceed in parallel. The IPDS simulation must be able to evaluate control strategies that are not yet defined and the ability to quickly respond to new requirements and system enhancements will continue to be crucial. Additional automated testing offers support for evolutionary refactoring as the system functionality expands and initial experiments with new specification and testing frameworks appears promising.

7 Conclusions

This paper describes the iterative design and construction of an architecture prototype for an intelligent power distribution system with the AO-MaSE process. It describes a recommended software engineering process employing specific design conventions that begin simply and focus on moving sooner from initial concepts to code construction while creating an evolving, iterative framework suited to the development of complex, adaptive, intelligent, autonomous systems.

The effort includes policy recommendations and detailed guidelines that produce a vertical slice of a complex system earlier in the process, forming a working core that enables quicker feedback into the behavior of a complex, recursive HMAS. Architectures that don’t perform as expected can be abandoned sooner and alternate versions tested. The process is compliant with the proven O-MaSE process and enables the full functionality needed for complex control systems yet offers a structured path towards implementation that addresses several challenges encountered when developing MAS. Specific recommendations are included with examples taken from the initial IPDS implementation.

8 Future Work

The prototype allowed us to test several critical issues early. The next phase will involve an initial assessment of potential self-organizing abilities, grid-based leader election algorithms, implementation of voltage/var control strategies during periods of renewable intermittency, initial critical power supply strategies, and reconnection approaches after islanded operation. The first PDS GUI will be created and agent negotiation and support capabilities will be tested between complementary lateral power lines in a selected IEEE test case. The AO-MaSE process will continue to be employed and refined to support a more adaptive implementation of evolving functionality.
9 References

Propagating AUML Protocols to Detailed Design

Yoosef Abushark * and John Thangarajah

School of Computer Science and IT, RMIT University, Melbourne, Australia
{yoosef.abushark,john.thangarajah}@rmit.edu.au

Abstract. The interaction between agents is a key aspect of multi-agent systems. AUML sequence diagrams are commonly used to specify these interactions between agents in terms of interaction protocols. Whilst most of the popular agent oriented software engineering methodologies such as Prometheus, Tropos, O-MaSE and GAIA support AUML protocol specifications in the design, the supportive tools do not provide any mechanisms for ensuring that the detailed design, and consequently the implementations, faithfully follow these protocols. In this paper, we show how AUML protocol specifications in the Prometheus methodology can be automatically propagated to the detailed design of the methodology by creating appropriate artefacts.

Keywords: AOSE Methodology; Multiagent system; Inter-Agent Interaction Protocols

1 Introduction

Intelligent Agent Systems are gaining popularity for building complex applications such as Unmanned Aerial Vehicles [19] and Electronic trading agents [21]. Features such as autonomy, proactivity, flexibility, robustness and social ability, are what makes these multi-agent systems (MAS) suitable for developing applications that operate in highly dynamic environments. However, these very features also makes developing and testing multi-agent systems a difficult and challenging task.

A number of architectures have been proposed to developing MAS, in particular, the popular Belief-Desire-Intention (BDI) agent architecture [20] where agents are developed using mental attitudes of beliefs, goals, plans, events, and so on. A number of agent oriented software engineering (AOSE) methodologies have been proposed for designing and implementing systems based on the BDI model of agency. Amongst them, Prometheus [16], Tropos[3], O-MaSE [7, 6] and GAIA[24] are some of the most commonly used.

In multi-agent systems inter-agent interaction plays a significant role. For example, in an agent-based trading system, the buyer and seller agents need to communicate with each other in order to complete a sale transaction. The above agent design methodologies allow the designers to capture these interactions

* Acknowledges King Abdulaziz University for scholarship
in the form of interaction protocols. A common representation of interaction protocols is AUML (Agent Unified Modelling Language) sequence diagrams [14], as adopted by the above mentioned design methodologies. An AUML sequence diagram captures all the possible legal exchange of messages between agents including the temporal aspects.

Although most of the AOSE methodologies consider agent interaction protocols an essential part of the methodology, they provide little (if any) support for ensuring that the interaction protocols are faithfully translated from specification to the detailed design artefacts. It is up to the designer to ensure that the protocols are indeed followed by the system, which can be a tedious and error-prone task that often result in a mismatch between the specification and implementation.

In this paper we present an approach to address the above. In most of the AOSE methodologies, the detailed design (the lowest level) is the closest to implementation and often can be auto-generated to skeleton code. In this work, we provide a mechanism for automatically creating these detailed design structures from the AUML protocol specification. We base our approach on the Prometheus methodology. For example, in the Prometheus methodology the development of protocols occurs in the ‘Architectural Design’ phase, but they do not get mapped fully to the detailed design phase of each agent which will be eventually translated to skeleton code in the JACK agent language [22].

In the following section, we briefly describe the AUML protocol specification and how it relates to the current AOSE methodologies. We then describe some of our propagation mechanisms in Section 3 including the factors that influence the algorithms. We then evaluate our approach in Section 4 and show that even for a simple protocol, relatively experienced users take a considerable amount of time to manually translate protocols to implementation and also produce a number of errors in terms of not adhering to the protocol specification. Our approach overcomes these drawbacks and provides a significant time saving as well as a more reliable system with respect to protocols.

2 Background

AUML sequence diagrams are a popular way of representing protocols and has been adopted by the popular AOSE methodologies. There are however other approaches such as the Finite State Machine approach used in the work on Electronic Institutions [1]. In this section we briefly describe the AUML protocol notation and the current support for protocol development in the common AOSE methodologies; Tropos, O-MaSE, GAIA and Prometheus.

2.1 AUML Protocol Specification

AUML is an extension of the standard Unified Modelling Language (UML) used in the object-oriented paradigm [14]. The purpose of AUML is to generate artefacts that support the development environment throughout the development lifecycle. Even though the AUML notation supports the entire development lifecycle, in this work we are only concerned with the AUML notation for modelling agent interaction protocols. More specifically, the AUML sequence diagrams.
AUML sequence diagrams [12], also called a Protocol Diagrams [2], is one of the dynamic AUML models that shows the flow of messages between agents and the order of those messages. Agents that implement the protocol must be able to send and receive the messages in the order specified. AUML sequence diagrams are similar to UML sequence diagrams that are used in the Object-Oriented paradigm. However, instead of having instances of objects as the main entities of the diagram, agents (or agent roles) are the main entities [15].

In addition to message flow, AUML sequence diagrams also allow constructs and guards to be specified. Constructs control the execution flow of messages specified and guards specify when a particular sequence of messages is valid (or not). The AUML sequence diagram has eight different constructs, as follows:

- **ALT** (alternative): can have multiple regions, with only one region that is executed based on the region’s guard (the condition that must be true for the region to be executed). It is possible that none of the Alternative’s regions get executed. To overcome such a situation, the ‘else option’ needs to be forced [12].
- **OPT** (option): is a single region that may or may not occur based on the guard of the construct.
- **LOOP**: indicates the repetition of a sequence of messages for a fixed number of iterations based on a number or a logical condition.
- **BREAK**: shows that the communication has been interrupted and terminated.
- **STOP**: indicates the end of the agent’s lifeline.
- **PAR** (parallel): allows the communication to be made in parallel.
- **REF** (reference): enables the designer to include another sub-protocol within the modelled protocol by referring to the name of that sub-protocol.
- **Continues [goto / label]**: is used to control the execution of the sequence of a protocol through two directives: ‘goto’ and ‘label’. The designer can make the sequence jump to a specific point within the protocol.

AUML sequence diagrams can be constructed in two ways; using either graphical or textual notations [23]. Figure 1 shows a simple AUML sequence diagram in both its graphical and textual notations. The figure illustrates the interaction between two player agents in a gold mining game. ‘Player:A’ agent asks ‘Player:B’ agent whether it has gold or not. The ‘Player:B’ agent may reply with either ‘Yes’ or ‘No’, based on the ‘Carrying Gold’ boolean predicate. Thus, the reply is embedded in an ‘ALT’ construct.

Whilst the graphical notation is an intuitive form to visualize the protocol, the textual notation is a structured way of constructing the protocol that is fast, easy to write and edit. In order to textually construct an AUML interaction protocol the written AUML textual notations must be well structured according to the rules specified in [23]. The Prometheus Design Tool (PDT) which supports the Prometheus design methodology allows the users to specify protocols using the textual notation and generates the visual diagram from it. We use the structured textual notation for implementing our protocol propagation techniques outlined in Section 3.
2.2 AOSE Methodologies and Protocols

In AOSE, there has been little research into ensuring the faithful implementation of protocol specifications. We consider here four of the most commonly used AOSE methodologies: Tropos, O-MaSE, GAIA and Prometheus. Despite the fact that these methodologies do offer development environments through their supported tools, none of them adequately support the propagation of the interaction protocols to lower design levels. We explore each of them below:

**Tropos Methodology:** In Tropos, the interaction protocols are specified in AUML as a part of the capability modelling activity in the detailed design phase. The Tropos methodology has many tools to support the methodology [13] and help with generating the design artefacts. One of these is the TAOM4e tool [7]. This tool has a code generation feature that takes a detailed design and provides skeleton code in the JADE agent language [13].

The UML2JADE code generator is used to generate JADE agent code with respect to the agent interaction diagrams. The JADE agent code is generated through the transformation from the interaction diagrams meta-model to the JADE meta-model that leads to the creation of a XMI (XML Metadata Interchange) file that helps to produce the capability files [18]. This meta-model propagation is limited to propagating the messages exchanged between the agents within the protocol but does not enforce the ordering specified in the protocols which is the key contribution of the work we present. Given that no ordering is enforced, the AUML constructs are also not considered in the propagation.

Further, the limited transformation is done directly from the AUML specification to code, rather than to the lower design levels, which precludes the designer from being able to modify and control the protocol elements at a design level. For example, a plan that sends a particular message in a protocol may perform other tasks that need to be modelled at the design level.

**O-MaSE Methodology:** O-MaSE consists of three design phases: requirements, analysis and design. Protocol development occurs in the design stage which has seven tasks[9]: model agent classes, model protocols, model plans, model policies, model capabilities, model actions and model services. O-MaSE uses AUML sequence diagrams for modelling the interaction protocols between agents[6].

The O-MaSE methodology offers a development environment for developing agent-based systems through agentTool III (aT³) [13]. aT³ provides a complete
code generation facility that produces an implementation skeleton code of the intended agent-based system according to the detailed models of that system [13]. However, it does not support the propagation of the modelled protocols to these detailed models [8].

**INGENIAS Methodology**: The INGENIAS methodology considers agent systems from five viewpoints: organisation, agent, goals/tasks, interactions and environment [17]. The designers are provided a set of concepts and relationships to describe each viewpoint in terms of design elements. These viewpoints represent the meta-models of the intended system. The definition of interaction protocols in this methodology is part of the interaction viewpoint designing process. Even though the methodology has its own notations (Grasia) for modelling protocols, it accepts AUML sequence diagrams to model the interactions [17].

The INGENIAS Development Kit (IDK) is an integrated development environment that supports the methodology’s development life-cycle [10]. The tool provides code generation capability that transforms the system’s meta-models into implementation code that targets the JADE platform [17]. Thus, the tool does not support the diagrammatic propagation of the design elements. Since the code generator can only identify the Grasia notations [11] and the IDK does not support the transformation of the AUML notations to Grasia, the propagation of AUML protocols is not supported in INGENIAS.

**GAIA Methodology**: The GAIA methodology enables agent designers to analyse and design an agent-based system through two main phases: the analysis and design phases [24]. In the analysis phase, the designers elicit all the possible entities of the intended system by using abstract concepts from the requirement statements. One of these entities is the roles that are needed in the system and the interaction between them [24]. Thus, two models result from the analysis phase: roles, which are later mapped to agents, and interaction models that capture the communication between agents in the system. In the design phase, the models derived from the analysis phases get detailed to a lower level of abstraction [24]. Three models are generated as the design phase’s output as follows: the agent model, the service model and the acquaintance model that defines the communication links between agents. For interaction protocol specification, GAIA also uses AUML sequence diagrams [4].

The GAIA for Eclipse designing tool (GAIA4E) aids the agent designers in documenting the activities of the methodology in terms of design artefacts [5]. The GAIA4E tool does not support the propagation of the created models, including the interaction models, in the earlier phases to the later phases of the methodology.

**Prometheus Methodology**: The Prometheus methodology consists of three phases: the system specification phase, the architectural design phase and the detailed design phase [16].

In the system specification phase, a translation of the problem that the intended system needs to solve is done based on the user requirements. Briefly, the requirements are taken as an input and the initial picture of the system is
drawn by defining the goals and the basic functionalities of the system. In this phase, the external entities (actors), system inputs (percepts) and system outputs (actions) of the intended system are defined. The primary output from this phase comprises two parts: system goals and scenarios.

The architectural design phase concerns the internal architecture of the system. Based on the system goals and scenarios from the previous phase, the roles and agent types of the system are determined. The system overview diagram captures the agents of the system, for each agent the events it handles and actions it generates, and the interaction protocols between those agents that communicate. Interaction protocols are modelled using AUML sequence diagrams.

In the detailed design phase, each agent type identified in the architectural design phase is designed in detail to fulfill its responsibilities according to the system overview diagram. Each agent has its own agent overview diagram where the agent is designed and detailed in terms of events, plans and belief sets.

The Prometheus Design Tool (PDT) [15] is a graphical tool that supports each phase of the methodology and provides designers with many features, such as visual editing, type safety, information propagation, report generation, cross-checking, and so on [15]. The code generation feature transforms the detailed design to skeleton code in the JACK agent language [22].

In PDT, protocols are specified using the AUML textual notation and AUML diagrams are generated by the tool. PDT currently supports the propagation of the agent and message entities in a protocol to lower level design diagrams. The propagation of the protocol trigger, the sequence flow and protocol constructs are however not supported. The onus is on the designer to manually map these elements to the detailed design phase which will then be translated to code. In this paper, we present an approach for propagating complete AUML protocol specifications to the detailed design phase of the Prometheus methodology.

3 Propagation Mechanism

We now describe the mechanisms for propagating protocols from the AUML specification to the detailed design in the Prometheus methodology. Although, we chose Prometheus as the target methodology the approach is applicable to any approach that follows the general BDI model. In our approach we consider simple AUML protocols without constructs and protocols that contain the ALT and OPT constructs (including those with multiple such constructs) as they are the most commonly used protocol constructs. We do not consider nested constructs at this stage which we leave as future work.

We begin by describing the factors that influence the propagation mechanisms and then describe the propagation mechanism by illustrating some examples.

3.1 The Factors

The automated protocol propagation task is to create the necessary design artefacts, that is, events and plans in the respective agent overview diagrams such that the sequence of message flow specified in the protocol (including constructs
if any) is adhered to. The detailed design is then translated to code. There are three factors that influence the protocol propagation to the detailed design: (i) protocol participants, (ii) protocol trigger, and (iii) protocol sequence flow.

**Protocol Participants:** The messages in a protocol are between two participants (internal agents or actors that are external to the system) and there can be many participants in a single protocol.

Where a participant is denoted as an agent in the AUML textual notation (see Figure 1 for an example) an ‘Agent’ entity is created in the System Overview diagram, if it does not already exist. The details of each agent, that is, the messages that it receives and sends, the data that it accesses and so on are detailed in the individual agent’s Agent Overview diagram.

In a protocol an agent can play two roles: Sender or Receiver for a particular message. The sender agent needs to be able to send the message and the receiver agent needs to be able to receive and handle (act upon) the received message. The message therefore needs to be propagated into both the agents together with a plan in the sender agent that sends the message and a plan in the receiver agent to handle the message. For example, Figure 2 shows the propagation of a single message from one agent to another.

**Protocol Trigger:** The protocol trigger is the event (possibly external) that triggers the posting of the first message of the protocol, thus initiating the execution of the protocol. It is important to factor this into the protocol propagation. It is often the case that the protocol trigger is captured by the agent that sends the first message of the protocol. However, in some cases it may be captured by more than one agent.

For example, in Figure 3 the first message (‘M1’) is sent by ‘Agent-A’, hence, the protocol trigger would be captured only by ‘Agent-A’. However, the first message of the protocol in Figure 4 might be ‘M1’ from ‘Agent-A’ or ‘M2’ from ‘Agent-B’ depending on the construct’s guards. Hence, both agents need to capture the protocol’s trigger with the guards propagated to the respective plans as context conditions that handles the trigger of each agent.

**Protocol Sequence Flow:** The sequence flow of an interaction protocol is the execution order of the communication between the participants. Thus, the propagation of the interaction protocols must ensure this sequence flow. When there are multiple messages, there are 3 distinct cases that influence the propagation mechanism: (1) Multiple messages sent in sequence from one agent, (2) Participants exchange messages and (3) Protocol contains constructs (ALT/OPT).

1. **Multiple messages sent in sequence from one agent:**
The first case is where an agent sends multiple messages to other agents continuously, for example, ‘M3’ and ‘M4’ in Figure 3. The significant point here is to ensure that ‘Agent-A’ posts these messages in the same order specified, in other words, ‘Agent-A’ must not post ‘M4’ before posting ‘M3’.

Currently in the Prometheus methodology (and PDT) there is no mechanism for specifying such an ordering, hence we introduce a new notation, a dashed-arrow, between the messages indicating the order of posting. For example, see Figure 5 which illustrates the propagation for the protocol in Figure 3. In ‘Agent-A’ overview diagram, ‘M3’ is posted prior to ‘M4’.

Note that, even if the messages were posted by different plans, we show the ordering of messages via dashed-arrows between messages rather than between the plans, as the protocol only specifies ordering of messages, not plans. Further, ordering the plans is too strict, unnecessary and possibly undesirable as the plans may contain steps other than the posting of the message and are often executed concurrently.

2. Participants exchange messages:

The second situation is when the protocol participants exchange messages between each other. For example, see the order of messages ‘M1’, ‘M2’ and ‘M3’ in Figure 3. In this case, it is important to ensure that ‘Agent-A’ sends ‘M3’ after receiving ‘M2’ and that ‘Agent-B’ sends ‘M2’ after receiving ‘M1’.

We enforce this ordering when messages are exchanged by having the plan that handles the incoming message post the outgoing message. For example, for the protocol in Figure 3, ‘Agent-A’ will have ‘M1’ posted by the protocol trigger handler plan as it is the first message of the protocol, and ‘M3’, ‘M4’ posted by the ‘M2 Handler’ plan. Similarly ‘Agent-B’ has a plan that handles ‘M1’ and
posts ‘M2’, ensuring that ordering (see Figure 5).

3. Protocol contains a construct:

The third situation arises when an interaction protocol contains a construct. In addition to enforcing the control specified in the construct (which we describe in the next subsection), having messages before and/or after a construct also affects the propagation mechanisms.

**Before:** If there are messages before the construct, then the last message before the construct is treated as the trigger for the construct which is created as an internal event. To illustrate this consider the protocol in Figure 6, which shows the protocol and the propagated ‘Agent-A Overview Diagram’. The plan that posts the message ‘M1’ will also post an internal event ‘ALT trigger’ to trigger the ALT construct. This internal event is handled by two plans with the guards of the ALT construct posting ‘M2’ and ‘M3’ respectively.

**After:** In the cases where there are messages after a construct, the propagation mechanism needs to consider the fact that the construct in some instances may not occur. For example, in the protocol specified in Figure 7, if the guards ‘X1’ and ‘X2’ both evaluate to false, the ALT construct will not execute and the message flow should continue past the construct.

Considering the direction of the first and last message of a construct’s regions, the occurrence of messages before and after a construct and the direction of those messages, provides many combinations of unique cases to be considered when propagating protocols (though, some of the cases are uncommon and unlikely to appear in practice).

In developing the propagation techniques we discovered nineteen unique cases for a protocol with just an ALT construct, and twelve unique cases for the OPT construct. Due to spatial reasons we do not attempt to describe all these cases nor the full details of the algorithms in this paper. However, in order to illustrate the propagation algorithms we step through a particular example with an ALT construct in the next section. We also highlight a situation where designer intervention is necessary. For a full list of all the different cases including cases where a protocol contains multiple constructs we refer the reader to a detailed appendix which we have placed online (anonymously) at http://tinyurl.com/propagation-cases. Similarly, the algorithms in the form of pseudo-code can be found at http://tinyurl.com/propagation-algorithms.
3.2 ALT Construct Example

The ALT construct consists of at least two regions, each with its own execution condition (region guard). Only one of the construct’s regions will be executed (possibly none). A region may have more than a single message within it, in which case the guard is applicable to the first message of the region. In this particular example, we consider the protocol shown in Figure 7 where the ALT construct is at the start of the protocol and there is a message that follows it. The complete Agent Overview diagrams generated by our propagation mechanism is shown in Figure 8.

The protocol propagation must consider the propagation factors discussed earlier: protocol participants, protocol trigger and protocol sequence flow.

The **protocol participants** are propagated by creating ‘Agent-A’ and ‘Agent-B’ in the System Overview diagram.

The **protocol trigger** in this case triggers the ALT construct as it is at the start of the protocol. Given that the first message of both the regions in the ALT construct is sent from ‘Agent-A’, the trigger is propagated to the ‘Agent-A Overview Diagram’ as follows:

- A ‘Protocol Trigger’ event is created.
- Three plans are created to handle the trigger event: ‘ALT Region#1’ plan, ‘ALT Region#2’ plan and ‘Else Option’ plan. The first two plans are given as context conditions the guards of the respective ALT regions. The last plan is created to handle the case where neither of the ALT regions execute, hence the context condition is the negation of the conjunction of the two guards (that is, !(X1&&X2)).
The sequence flow is then considered. First the ALT construct is dealt with as follows:

- First, plans for dealing with the two regions of the ALT construct and a plan to handle the case when the construct does not get executed needs to be created. However, these were already created when propagating the protocol trigger.
- Each region contains just one message, each sent from ‘Agent-A’ to ‘Agent-B’. Thus, messages ‘M1’ and ‘M2’ are added to the ‘Agent-A Overview Diagram’ and attached to the corresponding ALT region plans as messages posted. These messages are also added to the ‘Agent-B Overview Diagram’ as incoming messages and plans ‘M1-Handler’ and ‘M2-Handler’ are created to handle them.
- If the ALT construct does not execute (else-option) then the sequence flow is such that ‘Agent-B’ sends ‘R1’ to ‘Agent-A’. In order for this to occur, ‘Agent-A’ needs to notify ‘Agent-B’ that the ALT construct has ended. Hence, we add an event ‘End of ALT’ that is sent by the ‘Else Option’ plan in ‘Agent-A’ and handled by a plan in ‘Agent-B’.

Having propagated the ALT construct, we then propagate the message ‘R1’ to ‘Agent-B’ as sent by all three plans that signifies the end of the ALT construct and to ‘Agent-A’ as an incoming message handled by plan ‘R1-Handler’.

A Special Case: In the situation where the messages of the ALT regions are not sent in the same direction, for example as in Figure 9, forcing the else-option as done above presents the following challenge. The plan that enforces the else-option in ‘Agent-A’ for example, will have as the context the negation of both the guards. The guard ‘X2’ however, may be local to ‘Agent-B’ as the relevant message is sent from that agent. The designer therefore, needs to take this into consideration and ensure that the else-option plans do have access to the necessary guard conditions. To address this issue, the propagation algorithms create a note attached to the else-option plan that face this issue, as shown in Figure 9.

4 Evaluation

In this section, we perform a simple evaluation to validate that:

- manually propagating protocols from specification to detailed design and subsequently code can be a time-costly and error-prone task; and
– our automated propagation mechanisms significantly reduces these costs.

We do this by creating a simple system with a basic protocol specification and have it propagated to the detailed design and code in two ways: (i) by using the proposed automated approach and (ii) by using two human participants that are relatively experienced in using Prometheus and the JACK agent language. The first participant was a recent graduate who had studied agent programming and design, and had worked on projects across a year using PDT and JACK. The 2nd participant had 2.5+ years experience as a software developer using PDT and JACK. We then compare the relative costs and examine the correctness of the different solutions.

4.1 Experimental Setup

We developed a prototype 'eTrading-System' as a multi-agent system with three agents: 'Seller Agent', 'Buyer Agent' and 'Bank Agent'. We specified one interaction protocol, 'Sale Transaction' where the agents communicate with each other as shown in Figure 10. The system was designed in PDT up until the System Overview diagram. The task was then to complete the Agent Overview diagrams, auto-generate code, and complete the implementation. Completing the system code from the auto-generated code involves implementing: (i) the context conditions of the plan (ii) the body of the plan which posts the relevant messages, and (iii) the protocol trigger that initiates the protocol. The participants were given these instructions. Participants also tested and debugged their systems and finished when they were confident that their systems followed the protocol specification. In order to determine whether the automated approach does save development time, we observed the participants and recorded the following timing information:

– **Propagation time**: time taken for propagating the protocol to the detailed design phase (first draft).
– **Implementation time**: time taken for completing the code including the plans' contexts, plan bodies and protocol trigger.
– **Debugging time**: time taken for testing and debugging the completed system. This testing includes executing suitable test cases and when errors are
found, fixing the agent’s detailed design by adding and removing entities (events and plans). By error we mean, when the protocol is not followed. We also record here the number of iterations between code and design.

The implementation and execution of the systems were done locally on a client machine, without any external entities, such as servers. After all three systems were completed, we augmented the plans and agent code with log statements to keep track of the activity of the system.

### 4.2 Results

Due to spatial reasons we do not attempt to show and analyze the detailed design of the agents produced in each of the three systems, however, in Figure 11 we present the Agent Overview digram of the ‘Seller’ agent as detailed in the three systems.

Table 1 shows the time-cost for developing the 3 systems. As evident from the results, even for the relatively simple protocol with just two constructs, there is a significant saving in development time when the automated propagation is used (over three hours of savings for even an experienced agent program developer like the 2nd participant). For a system that contains a number of protocols, including more complex protocols than our test system, the time savings would indeed be much greater.

The marked difference is in the ‘Propagation Time’ and ‘Debugging Time’. The propagation time is instantaneous using the automated approach, whilst the manual propagation by the human participants took over an hour for each.

<table>
<thead>
<tr>
<th></th>
<th>Automated</th>
<th>1st</th>
<th>2nd</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Propagation Time</strong></td>
<td>Instantaneous</td>
<td>75 minutes</td>
<td>50 minutes</td>
</tr>
<tr>
<td><strong>Implementation Time</strong></td>
<td>30 minutes</td>
<td>64 minutes</td>
<td>80 minutes</td>
</tr>
<tr>
<td><strong>Debugging Time</strong></td>
<td>25 minutes with 0 iteration</td>
<td>130 minutes with 5 iterations</td>
<td>120 minutes with 3 iteration</td>
</tr>
<tr>
<td><strong>Total Time</strong></td>
<td>55 minutes</td>
<td>296 minutes</td>
<td>250 minutes</td>
</tr>
</tbody>
</table>

**Table 1. Time costs**
The debugging time when using the automated approach is for testing the system. As the tests did not result in any errors, there were no iterations between code and design. This was not the case for the manual propagation by the test participants.

Note that, neither of the propagation time nor the number of iterations between code and design for debugging will increase as the number of protocols increase when using the automated approach. However, they will increase (on average linearly) with the number of protocols when manually propagating.

We see that there is also a significant saving (more than 50%) in ‘Implementation Time’ when using the automated approach. This is due to the fact that the protocol trigger and context conditions of plans are propagated to the detailed design from the protocol specification. The overhead for the test participants was to figure these aspects out.

### 4.3 Error analysis

To test whether the implemented systems did indeed follow the protocol specification, we ran each system with 6 different test cases that represent the 6 different sequence flows as specified by the protocol. By following the activity logs and cross-checking with the protocol specification we determined if there were any errors in the test execution and recorded these errors. We considered the following as errors in the interaction:

- if any message that is expected to be sent is not sent.
- if any message that is expected to be sent once gets sent many times.
- if any message sent is not handled.

Table 2 shows the total number of errors detected in testing the three systems. As shown, the system that followed the automated propagation mechanisms produced no errors, which validates the correctness of the propagation algorithms for the protocol specified. The systems developed by the test participants however, produced 18 and 6 errors, respectively. As with the time-costs we would expect these errors to increase with an increase in the number of protocols and their complexity.

**Cause of errors:** The first participant made errors in handling the sequence flow between the OPT and ALT construct and also did not implement some of the messages in the protocol as messages.

His implementation was such that after the ‘Give-Price’ message is sent, the ALT is always triggered. The ALT is also triggered after the OPT gets executed, thus executing the ALT construct twice (resulting in 6 errors). In our proposed

---

1 There may be exceptional situations where protocol specific debugging is required.
approach, we avoid this issue by creating an event that triggers the OPT and ALT constructs appropriately (similar to what is described in Section 3.2).

There were also two messages (‘Accept-The-Price’ and ‘Refuse’) that were not implemented as messages in the system, instead they were embedded as business logic in plans (resulting in 12 errors).

The more experienced second participant also made a few mistakes. His implementation resulted in the message before the OPT (‘Give-Price’) and the OPT construct being exclusive. This meant that only one of them ever got executed, even though, both should be executed in the case where the guard of the OPT is true. In our proposed approach the plan that posts the ‘Give-Price’ message also posts a trigger for the OPT construct which is handled by a plan that gets executed only if the guard of the OPT is true.

We note also that there were several mismatches between the design and the code for both participants.

5 Conclusion

Interaction between agents is a central aspect of multi-agent systems. AUML sequence diagrams are a popular means for specifying these interactions and is the adopted model in current AOSE methodologies. The lack of support for propagating these protocols to more detailed design level of these methodologies, means that the onus is on the designers to ensure that the protocols are faithfully implemented. In this paper, we showed empirically that this manual propagation of protocols is a time consuming and error-prone task even for a relatively experienced agent system programmer.

To overcome the above, in this paper we proposed an approach for automating the process of propagating the protocol specifications to the detailed design levels in the Prometheus methodology. We identified the factors that influence the propagation, the different cases that the mechanism needs to consider and the algorithms that performs the automated propagation. This approach can be extended to other AOSE methodologies that follow the BDI model of agency.

The evaluation that we conducted showed that by automating this process, the development time is significantly reduced and the resulting system is more reliable than the manual process. Whilst the evaluation was not comprehensive in terms of the number of participants and variation of protocol specifications, it provided a good indication of the kind of benefits that our automated approach provides and the difficulty of the manual propagation. We note that, one of the participants was an experienced programmer and hence envisage novice programmers to encounter much more difficulty. Further, the test system contained a single protocol, with just two constructs. It is fair to assume that a larger system with many more protocols, some, more complex, will see a greatly increased benefit in adopting such an automated approach.

References

Abstract. Traditionally, when specifying system requirements, analysts cut the solution space in order to define the expected system behavior in a deterministic way. However, the characteristics of being autonomous and proactive make the agents able to explore a wider solution space, even when this space dynamically changes or contains uncertainty. We propose a language for describing system goals that may be injected at run-time into the system. The novelty of our approach consists in making less rigid some constraints of the solution, thus allowing more degrees of freedom to the system. On the occurrence, agents may exploit their features (mainly autonomy and proactivity) for benefiting of the freedom degrees of the solution space. In this way, agents may adapt their behavior to the current operating conditions. In addition, this approach implements an efficient mechanism for the evolution of system behavior because the expected behavior may be changed during the system life-cycle by modifying the injected set of goals. This paper focuses on the goal specification language, and discusses how it supports the adaptivity and the system evolution.

1 Introduction

The current work arises in the context of the project Innovative Document Sharing (IDS)\(^1\), whose aim is the development of an adaptive and autonomous workflow enactment engine for improving task coordination and document management in small and medium local companies. The project exploits the well-known BPMN standard [1], among its assets, because the system will be used in business contexts. Indeed the BPMN is mainly targeted to humans, being very flexible and expressive. In addition, the BPMN standard includes the notation to describe workflows as orchestration of both automatic services and human tasks. On the other side, the business domain is highly variable application context. Business rules could change very frequently due to evolving business strategies, changing of company short/middle term goals, or due to the dynamic society, with its laws and regulations that must be respected. BPMN does not support this dynamic context. Every external change must be implemented into the workflow as a set of modifications. It follows that the workflow must be re-designed for implementing the new requirements, but checking inter-dependencies and verifying the validity of the result.

\(^1\) The IDS project is funded by the Autonomous Region of Sicily (POR FESR Sicilia 2007-2013)
It is a matter of fact that evolving business model is not a trivial task: a great number of malfunctions in workflow systems depend on business analysis errors [2–4]. These are the main motivations to adopt a workflow system able to autonomously react to changes of the context where it is working on [2]. An adaptive workflow behaves like a normal workflow and, moreover, is also able to react to some changes in the environment [2]. The need for self-adaptation is often linked to the need of react to exceptions [5]. BPMN [1] already provides mechanism to specify how the system will react to expected exceptions. It is more and more interesting to define how to react to unexpected exceptions, which are events that cannot be handled by the main process flow. Another important manifestation of adaptivity is the ability of individual organizations and users to customize their software to their unique and changing needs in different situations and contexts [6]. Therefore, self-adaptation is a desirable feature for a workflow system. Indeed it is also widely recognized that workflow management systems should provide flexibility [2–4].

The concept of self-adapting system is gaining more and more attention in the area of agent-oriented software systems. Multi-agent systems encapsulate the adequate level of abstraction that is needed to implement software system capable to react to changes. Adaptivity grounds on autonomy since the system must be able to modifies its behavior when it is necessary. Adaptivity requires the ability to monitor parameters of the context that may variate, and to reason on the perceived values in order to apply the appropriate strategy. More recently, an additional agent feature is becoming more and more central to self-adapting systems: the self-awareness. The agent that is aware of its design goals may reason and act in order to address them, even when the context changes. The specification of these design goal (and therefore of system requirements) plays a central role in defining self-aware agents.

Traditionally, requirements specification languages are commonly used for reducing the ambiguity of models, by limiting the solution space to a unique solution or a finite set of solutions for a given application context and/or rule set. This choice is not adequate for designing self-adaptive systems because it does not allow systems to modify their behavior according to changes in their environment [7]. A system is self-adaptive when it is able to autonomously decide an effective direction in a wider solution space in order to obtain the desired result. The goal-oriented approach, on the other side, allow to define the behavior and the expected result by means of a goal set and their relationships.

The proposed approach consists in considering a system goal as a component of the agent environment, an entity that evolves in time. Indeed, the designer may specify system goals at design-time or at run-time to make the system evolve.

Since an adaptive system react and adapt itself at run-time to changes of its environment, if the environment includes system goals agents will be able to acquire goal changes and to act as a consequence. We are planning to implement a workflow engine as a multi-agent system where agents are autonomous and proactive entities able to reason and act (as usually agents do) but also aware of the dynamic mission of the whole system. Self-awareness is fundamental but it grounds on an adequate goal specification language. Several approaches exist in literature to specify goals, but to the best of our
knowledge none of them is completely suitable for our scope. Therefore, we developed a goal specification language, namely GoalSPEC, that meets to our needs.

The remaining of the paper is organized as follows. Section 2 describes the motivation of our work, and Section 3 presents a short review of existing goal-oriented specification languages. Section 4 describes the characteristic of GoalSPEC. Some discussions and final conclusions are presented in section 5.

2 Motivation

The IDS project aims at developing a workflow enactor system for improving task coordination and document management in small and medium local companies. The project is a benchmark for exploring real motivations since a workflow system shall autonomously configure its behavior. In particular the requirements elicitation phase of the project highlighted that the business context is a high dynamic context, in which company business goals and rules often vary. In addition the workflow engine runs in a socio-technical context in which the human and social factors are relevant. In particular such systems operate in a society with laws and regulations that are frequently revised and modified. Nowadays the specification of business processes is done by well assessed language such as BPMN [1] or BPEL [8]. Let us consider the business process in Fig. 1 as an example of BPMN diagram that models a workflow for book management in a public library [1]. The workflow includes some automatic tasks (Get Book Status, Request Hold), a manual task (Checkout Book), and a set of send/receive tasks used to communicate with customers. The workflow starts when a customer requests for a book. If the book is available the library clerk checks out the book and replies that book is ready. Otherwise, if the book is already loan, the workflow waits for one among the following events: (i) the customer declines the request, (ii) the customer holds the request, or (iii) customer does nothing for one week. Alternatives (i) and (iii) terminate the workflow, whereas path (ii) generates a loop, in which the system is waiting for two weeks and after that it checks again whether the book is available.

Modeling a workflow requires that business analysts acquire deep knowledge of the domain of interest. The better is the knowledge the more complete is the specification of the business process, that will also include possible exceptional cases or possible faults. Indeed, BPMN includes a notation for dealing with exceptions and for specifying how the system can recover them. Of course when the workflows is under-specified nothing can be done to identify and avoid or recover unexpected situations. Many recent works [4, 9–11] face with self-adaptive workflow engines because they would be a great value for enterprises. The objective of self-adaptive workflows is to make the enactment engine able to recognize anomalous situation that are not included in the specification. The most advanced visions of the adaptive workflows include mechanisms to proactively modify the flow of activities but granting the correctness of the result. The current challenge is the definition of a flexible mechanism for the workflow to replan its activity for overcome the unexpected situation. A promising approach in literature is to incorporate multiple strategies into the system design and to let the system to select the appropriate one that address the desired goal [6, 12]. Indeed, a goal may be generally addressed in many alternative ways, each with different trade-offs. A
representative approach [6] is that of modeling goals in a hierarchy that describes the expected outcome of the system. This goal model is created at design time and then each goal is instructed with the necessary implementing code which execution addresses the target goal. The requirements elicitation of the IDS project revealed us that just like in many other cases, the business context changes very frequently. The objective of this paper is to decouple business goals and their implementation. The business goals specify the expected results, whereas it is up to the system to autonomously and proactively decide how to address these results. In addition, by decoupling goals and activities we can dynamically inject/retreat goals into/from the system on the occurrence. Neither BPMN nor BPEL include any explicit specification of business goals (even if some attempts to create a Business Motivation Model (BMM) are ongoing [13]). Therefore, we require a goal specification language with the following characteristics.

1. The language shall be powerful enough to represent requirements and constraints for information systems. System requirements describe the expected results of the system and they can be articulated into functional and non-functional requirements. Whereas functional requirements describe the behavior in terms of the expected functionalities, non-functional requirements generally describe the expected performances of the system. An important aspect to consider is the massive presence of social factors into the requirements: system constraints are norms that specify what the system is obligated to do and what is forbidden.

2. The goal specification shall describe the expected behavior of the system. The focus must be on ‘what’ is expected so the language shall be in a declarative fashion. It is out of the scope of the language to describe how to address the specified goals. Moreover, describing system goals means to describe the expected result in terms of states of the world. Hence, the specification language shall be grounded on ontological bases.
3. The language shall be automatically interpreted at-run time by system agents. Agents are the main consumers of the goal specifications, because they are responsible to adapt their behavior to the injected goals. This means that, for each goal that is injected in the system, agents must be able to understand and translate it into a personal set of believes and then reasoning on them for deciding how to contribute to goal achievement.

4. The language shall be attractive for a business audience. It must be handled by human operators, simple to learn, understand and use. Business processes generally run in environment with a massive influence of human factors that are determinant for the system behavior. Humans and the society are the main producers of requirements and, to reduce the gap between human and their requirements, we want to adopt a specification language accessible to non-technical people.

5. The language shall be flexible enough to include points of uncertainty in specifications. As previously discussed, we want to increase the degrees of freedom of the solution much more than traditional specification languages do. The rationale is to allow the adaptation mechanism to move into a wider space where more solutions are possible with different trade-offs. The presence of more alternatives enable selecting the efficient solution for the current operative context.

6. The goal specification shall enclose the expressiveness of the BPMN. Business Process Modeling and Notation is a lower bound for the expressiveness of the language to select. The language must be able to cover the specifications of every business process that is represented in BPMN.

To select the most appropriate language for our purposes we have conducted a systematic review of the state of the art in goal specifications (see next section).

3 Review of Goal Specification Languages

Goal-oriented approaches are widely used in requirements engineering mainly to fill the knowledge gap between analysts and stakeholders. However, the concept of goal is employed during the several activities from requirement analysis to software implementation for different purposes. We conducted a systematic literature review, according the principles of Wohlin et al. [14]. In this review, the research question concerned the way goals are represented and if they match with the characteristics defined in the previous section. The table shown in Fig.2 summarizes the results of this comparison.

We identified 20 among the most relevant papers in the area, that are distributed in informal, semi-formal and formal approaches.

Informal approaches commonly express goals by using natural language text, which is the most commonly used in human communications. This type of goal representation presents a high degree of ambiguity. We discarded informal approaches in our review because they are too much distant from our purposes.

Semi-formal representations are the most frequently used techniques for specifying goals. They are based on a mixed graphical and text based notation, which facilitates the exchange of knowledge with stakeholders. Among semi-formal approaches can be also individuate two sub-categories: the structured and the hybrid approach. Structured languages has been proposed as an alternative to formal assertions. A structured language,
in fact, is formal enough to allow for automatic goals manipulation, but maintaining simplicity and flexibility of a natural language. Hybrid approaches instead combine diagrammatic with techniques more inclined towards mathematical languages and theories. In the following we report some examples of semi-formal approaches.

**Business Motivation Model (BMM)** [13] is a meta-model and a standard for capturing business requirements. BMM focuses on capturing semantically rich requirements that are useful for business analysis, querying, impact analysis, change management, and business reasoning. Its purpose is to specify business requirements in order to highlight "why the business wants to do something, what it is aiming to achieve, how it plans to get there, and how it assesses the result". Nevertheless, BMM does not come with a standard graphical notation, it has a broader scope than just goal modeling and therefore it has too many concepts (some of which are unclear or overlap with each other), it has no strong formal basis and does not address at all goal analysis and reasoning issues.

**Goal Modelling Language (GML)** [15] is founded on an abstract syntax defined in a meta-model and concrete syntax in term of a graphical notation. The concrete syntax is aligned with existing notations for goal modeling (such as Tropos). The abstract syntax instead ground around five main constructs (goal, actor, resource, constraint and behavior) and some relationships. Goals they handle can be organizational and human actors goals. GML differentiates goals in soft and hard goal. But hard goals must be measurable and norms must be specified in relation with them. As concerns relationships, GML also support: Access representing the access of behavioral relationship concepts to resources; Assignment that reflects the ownership of a goal, resource etc...; Constrain controls/influence/limits defines some aspects of the behaviors or resources. GML models are thought also to can be transformed into the BPEL specification.

Both BMM and GML do not own many of core features we are interested. Among the most important issues, they do not ground on ontological bases and not support reasoning with uncertainty.

As concerns structured languages, two exemplifications are **Goal-Scenario Coupling** [16] and **Goal Question Metrics (GQM)** [17]. Goal-Scenario Coupling [16] approach expresses a goal as a clause with a main verb and several parameters, where each parameter plays a different role with respect to the verb. In Goal Question Metrics (GQM) [17], instead, goals are defined by filling in a set of parameter values in a defined template. Template parameters refer to the purpose (object analyzed for the purpose of), the perspective (in term of who is interested and with respect to) and the environment (in terms of the application context). Representations of this type may still contain ambiguities. We cite these languages only for completeness because they are very distant from our purposes. Goal Scenario Coupling does not deal with system goals, at all.

Among the hybrid approaches, instead, there are Tropos/i* [18] [19], **Goal Requirements Language (GRL)** [20] [21] and Morandini at al. [12].

Tropos [18] [19] incorporates goal modeling concepts through the phases of software development. In particular, Tropos language (based on i*) grounds around some core concepts such as: goals (hard goals and soft goals), dependencies, plans (ways for fulfill a goal), resources, And/Or Decomposition relations, means/end relationship (means to reach a goal), and contribution relationships (expressing positive or negative contributes to goal achievement). Moreover, specific labels enrich the semantic of the
model. But in Tropos, the provided goal representation do not contain enough information to be automatically processed and assigned to agents. Moreover, Tropos goals do not ground on ontological bases and do not support uncertainty.

GRL [20] [21] is based on i* and NFR [22]. It is a language for supporting goal and agent-oriented modelling and reasoning about requirements, with an emphasis on dealing with non-functional requirements (NFRs). In GRL, a goal can be either a business goal or a system goal. A business goal express goals regarding the state of the business affairs the individual or organization wishes to achieve. System goals describe the functional requirements of the information system. GRL is a language more suitable for the first phases of analysis. The goal specifications it provides are not suitable for agents. Moreover it not support adaptation and uncertainty factors.

Morandini at al. [12] provide an extension of Tropos for self-adativity. It enriches goal models by specifying goal achievement conditions in relationship with the environment, and the possibility to model faults and corresponding recovery activities. They also consider adaptivity in term of behavior to be adopted by the system in order to address the environmental changes and to properly achieve its main goal(s). Morandini at al. set system goals as invariants. Variation points of the global behavior are granted by decomposing the main goals into trees of alternative sub-goals. The system uses advanced decision techniques to select among many alternative strategies to address the main goals. Moreover, the defined goals can be directly mapped to Jadex goals. This technique of designing expected exceptions is already comprised into the BPMN specification. However, the main limitations are that goal models are defined at design-time thus they can not be injected into the system at run-time and there is a strict coupling between goals and implementation.

Formal approaches introduce rigorously defined ad hoc languages, commonly in the form of assertions. This kind of approaches provide consistent, unambiguous and precise representations of goals. They also provide the basis for sophisticated tools that can handle automatically goal-based activities since they have well-defined formal semantics. But it is worth to underline that such formal representations need more complex reasoning procedures and more effort for specification. Moreover, they often result in a lower usability in industrial context, because of their strong formalism. Examples of formal goal specifications are provided by KAOS [23], Formal Tropos [24].

KAOS [23] consists of a formal framework where goals are defined by means of real-time linear temporal logic. Goals are classified according to some patterns (Achieve, Cease, Maintain, etc...) and categories (Satisfaction goals, Information goals, Robustness goals, etc...). The strengths of KAOS are a solid formal basis and powerful support for goal analysis and reasoning. Moreover, temporal logic is hardly executable and it requires analysis with a solid background in discrete mathematics and formal logic and it is not human oriented.

Formal Tropos [24] is a formal language that offers all the primitive concepts of i* [18] enriched by temporal specifications. As well as Tropos do, Formal Tropos describes all the relevant objects of the modeled domains along with their relationships. But the description of each object is based on two layers. The outer layer defines the structure of the instances along with their attributes. The inner layer describes constraints on the lifetime of the object in a first-order linear-time temporal logic. It allows to perform
<table>
<thead>
<tr>
<th>Represent.</th>
<th>KAOS</th>
<th>TROPOS</th>
<th>Formal Tropos</th>
<th>GQM</th>
<th>GRL</th>
<th>Goal Scenario Coupling</th>
<th>Tropos for adaptivity</th>
<th>GML</th>
<th>BMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semi-formal</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Goal Type</th>
<th>Analysis</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Adaptivity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Features</th>
<th>Requirements/Constraints Representation</th>
<th>x</th>
<th>x</th>
<th>x</th>
<th>v</th>
<th>x</th>
<th>v</th>
<th>x</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ontological Representation</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Context-free Grammar</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Uncertainty Encapsulate</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Human Oriented BPMN Mapping</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>v = partial matching</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

**Legend**

Table 2. This table presents a comparison among goal languages and features we are interested.

verification/adaptation by analyzing huge number of scenarios with the purpose to find those which may lead to system failure. The formalization of goal models in temporal logic is an approach which may become unfeasible for large, complex systems. Instead, we want to model goals for adaptive systems in a such way that the system is able to find the appropriate adaptation process at run-time and to reorganize itself when requirements changes. In addition, *Formal Tropos* does not support BPMN mapping.

Fig. 2 summarizes the results of the review. This table provides a kind of matching between the analyzed goal languages and the requirement we need. At the best of our knowledge, we have not found one approach that fully meets all our requirements. In particular, the attempts to use Goal-based modeling for specifying adaptation do not fully satisfy what we want to realize in our envisioned framework. Therefore, the next section proposes a new language, named *GoalSpec*, that incorporates some features of the reviewed languages but also it introduces new characteristics for our purposes.

## 4 GoalSPEC: A Language to Specify System Goals

Here, we define a language for specifying system requirements and constraints that wants to be general enough to cope with several aspects that are key elements in current systems. To do this, we have incorporated some features of existing languages and we have introduced new ones in order to address specific adaptation and business issues. For the sake of clarity we prefer to refer to the whole language as an abstract package that contains two sub-languages: *GoalSPEC* that focus on specifying expected results...
of the system in terms of functions, and *NormSPEC* that is a norm-based language for specifying non-functional requirements and constraints that generate a boundary where the system is limited to move. This paper focuses on GoalSPEC only, whereas the foundations of NormSPEC are yet published in [25].

The common characteristics of both GoalSPEC and NormSPEC are: (i) their grammar is a subset of the natural language, (ii) they have context-free grammar, thus to be automatically parsed and translated into machine instructions, (iii) some elements of the specifications can be relaxed by using fuzzy modifiers.

The concept of system goal is central in our language. Aligned with common definitions in literature we distinguish between business goals and system goals.

**Def.** Business Goals are enterprise strategic interests that motivate the execution of business processes [26]. They are discovered in phase of analysis and are useful to model a strategic view of domain stakeholders and to elicit system requirements.

**Def.** System Goals are described as states of the world that the system desire to achieve [19]. System goals are generally the subset of business goals that are delegated to the workflow system in order to implement some kind of automation.

In the following, an extract of the BNF description of the GoalSPEC language:

```
<goal> := when <condition> <actors> shall address <state>.
<condition> ::= <event> | ( <condition> ) | <condition> <op> <condition>.
<state> ::= <predicate> | ( <state> ) | <state> <op> <state>
    | message <msg> is sent to <participant>.
<event> ::= on <date> | after <time> since <condition>
    | when <msg> incomes from <participant> | when <state>.
<msg> ::= <state>.
<actors> ::= <actor> | <actor> AND <actor> | <actors>, <actor>.
<actor> ::= the role <qualifier> | the system.
<op> ::= and | or.
```

**Social goals and agent goals.** The language productions allow to specify system goals. The first production of the BNF describes a goal as composed by an initial triggering *condition*, a list of *actors* that are involved and a desired final *state* of the world. We consider two categories of system goals:

- **social goals** are collective goals that specify top-level goals which achievement may be obtained by addressing lower-level social goals or agent goals. Therefore, social goals are related to generate a collaboration among many agents.
- **agent goals** are related to generating a specific outcome in the workflow; so they generally can be addressed by a single agent of the system. Generally a system goal exists to contribute to the achievement of a social goal.

The *actors* section specifies `who` is the main responsible to address the given goal. GoalSPEC includes two different categories of workflow participant: human roles and the system. When *the system* is the main actor of a goal, it means the goal may be automatically addressed. Conversely when a human role is responsible of a goal, the system can only monitor when the goal is successfully addressed. When the list of actors counts more than one participant, then the goal is social and it describes a collaboration among many parts.

**Triggering Conditions.** Each goal specification starts with a set of *conditions* that must hold because the goal is active. The BNF specifies that a condition may be a single event or a composition of multiple events. Basic events may be:
- on <date >, generates an event when a given day arrives. The date is a parameter that follows the ISO 8601 [27] (International standard date and time notation). Examples are 'on 1995-02-04', or 'on 2013-04-01/23:59:59'.
- after <time> since <event >, generates an event after an amount of time since a given past event occurred. The time parameter specifies a duration of delay according to the ISO 8601 standard. For instance 'after 2W' means 2 weeks.
- incoming <msg >, generates an event when a given message incomes. A message is specified as a content and a sender. When a message that matches with specified parameters incomes then the event is thrown.
- when <state >, generates an event when a specified state of the world becomes true. Later in this section the specification of states is explained in details. An example is 'when available(Book)'.

For instance, the triggering condition of the GetBookStatus goal (Fig. 1) is:

WHEN book_request(Book) INCOMES FROM Client OR
AFTER 2 weeks SINCE WHEN hold_request(Book)

It is up to agents to monitor their environment in order to catch when each single condition applies: the 'incoming of the book_request(Book) message' or 'two weeks are past since hold_request(Book) is true'.

States of the World. GoalSPEC adopts an ontological description of the world, and logic predicates play a central role in specification of elements and relationships. The BNF specifies that each goal specification includes a state that must be true in order to declare the goal is satisfied. A state may range from a single logic predicate, a composition of multiple predicates or sent message, when a message is sent to a participant. We selected Prolog as the dialect to define first-order logic predicates in a declarative fashion. This decision has been done also to be compliant with some BDI (belief-desire-intention) frameworks such as the Jason architecture [28].

Prolog predicates use atoms to represent: (i) particular individuals or objects (symbols starting with lowercase, or numbers); (ii) variables (symbols starting with uppercase) that will assume a value with the mechanism of unification; (iii) facts (functors followed by list of arguments), used to represent properties or relationships. Let us consider the information that the requested book is available in the library. This information may be represented in symbolic form by the predicate: available(Book) where Book is a variable that will contain the ID of the requested book. The predicate will assume a value (true or false) after the variable assumes a value. The atom available is a property that we want to check for the variable value. An example of relationship is request(Book, Client) that will assign the intention of a given Client to borrow a Book from the library. For instance, the final admissible state of the world of the GetBookStatus goal (Fig. 1) is a composition:

available(Book) OR loan(Book)

It is up to agents to be aware of the capability necessary to leave the state of the world in one of these two facts.

Human participants. Business processes describe sequences of operations. The BPMN standard (in contrast with BPEL) allows to declare a human role as responsible of activities. Manual and user tasks are operations that are executed without (or
with a limited support) of the machine. In our vision all the activities in a workflow, including the manual ones, exist in order to pursue a business goal, or in technical terms, to take the world in a desired state. Whereas goals that derive from service tasks are clearly assigned to one or more agents of the system, goals of manual/user tasks are addressed by human resources. The role of the system, in these situations, is to monitor that the desired state is correctly addressed, or when this is not feasible, to generate user interfaces where operators may notify their progresses. For instance, after the manual activity CheckoutBook (Fig. 1) the clerk fills a form in a web application to inform the system that the requested book is going to be lent to a given client.

**Fuzzy modifiers.** It is very interesting the work by Whittle et al. [29] who defined RELAX, a language for requirement that uses a declarative style for specifying possible sources of uncertainty. Flexibility is obtained by relaxing the rigid 'shall' form typical of requirements and by introducing uncertainty factors. RELAX is based on three types of operators (temporal, ordinal and modal) to address uncertainty. The semantics of RELAX expressions (AFTER $\varphi$, AS EARLY AS POSSIBLE $\varphi$, etc...) is formalized in terms of fuzzy branching temporal logic. The authors suggest that requirements languages for self-adaptive systems should include explicit constructs for specifying and dealing with the uncertainty inherent in self-adaptive systems.

Likewise in RELAX [29], the expressiveness of GoalSPEC language is extendable to let the designer to relax some constraints. This is possible by using 'fuzzy modifiers' to increase the flexibility of the rigid unification operator. For instance, it is possible to relax time constraints by specifying that a goal shall be addressed AS SOON AS POSSIBLE, or AS LATE AS POSSIBLE. This is particularly useful when the system deals with many parallel goals and must optimize the global behavior by selecting the one with highest priority. It is also possible to relax measures with the following modifiers: AS CLOSE AS POSSIBLE, AS MANY AS POSSIBLE, AS FEW AS POSSIBLE. For instance it is possible specifying the number of items in a list must be as close as possible to a given threshold. A value that is close to the threshold (but not equal) will not raise an exception, as a traditional workflow engine would do, thus allowing the workflow to continue normally. The implementation of these modifiers is a work in progress and is out the scope of this paper.

### 4.1 Translating BPMN into System Goals

Many times in this paper we mentioned BPMN [1] and BPEL [8] as the most common specification languages for workflow. In the IDS project we adopted BPMN because of its capability to describe human tasks that BPEL does not support. BPMN and GoalSPEC are not in competition, but rather are complementary. Each of them has a different role in the whole architecture. BPMN is to be the main interface for business analysts to model their business processes. GoalSPEC is intended to model the business goals above the BPMN process, but that are not explicitly expressed in it. The problem is that GoalSPEC must not be a further burden for the analysts in order to be effective. For this reason we developed a Java component that translates the BPMN workflow specification (XML) into a system goal set according the GoalSpec grammar. The main algorithm is Algorithm 1. It is worth to premise that any process or sub-process (a compound task) produces a social goals, whereas each relevant activity produces an agent
### Algorithm 1: Extracting Goals from the Workflow

Data: the BPMN workflow graph

Result: set of GoalSPEC goals

forall the $A$, activity in the graph do

  if $A$ is not a "receive activity" then
    generate a new goal $G$;

  forall the $I$, where $I$ is a predecessor activity of $A$ do
    search backward for the "triggering-event" $E$;
    concatenate $E$ with "conditions" of $G$ by the OR operator;
  end

  forall the $O$, where $O$ is a successor activity of $A$ do
    search forward for the "resulting-state" $S$;
    concatenate $S$ with "states" of $G$ by the OR operator;
  end
end

Inferring the triggering events of a goal. Given a workflow activity $A$, the triggering events of the corresponding goal $G(A)$ is the set of pre-conditions that let the execution of $A$. To extract these conditions, the sub-routine looks at all the incoming sequence-flows of $A$. Then, by looking backward these flows, it search for the events that any predecessor node $Pre_j(A)$ generate as exit conditions. For instance "GetBook-Status" has two predecessor nodes: "ReceiveBookRequest" and the timer event. Clearly, predecessors produce different exit conditions, depending if they are task, gateway or event nodes. Tasks port-condition is a change in the state of the world. Gateways do not generate any event, but rather they propagate the events in input. Finally, each specific event node produces a different kind of exit condition (incoming message, timer, compensation and so on). For instance, the goal related to 'GetBookStatus' activity has two alternative triggering events:

```
WHEN book_request(Book) INCOMES FROM Customer --> from ReceiveBookRequest
AFTER 2 weeks SINCE WHEN hold_book(Book) INCOMES FROM Customer --> from the timer
```

In this case the whole goal triggering condition is an OR composition of two events. Another example is the 'CheckoutBook' task whose condition depends on a state of the world: WHEN available(Book).

Inferring the resulting state of a goal. Given a workflow activity $A$, the resulting state of the corresponding goal $G(A)$ depends on: 1) eventual data-objects that are produced or modified by $A$, 2) the expected input of the successor node $Succ_j(A)$, 3) eventual alternative outgoing paths of $A$. The procedure looks forward at all the successor nodes $Succ_j(A)$ and tries to identify these cases. For instance the 'GetBookStatus'
activity is followed by an exclusive gateway that selects alternative flows depending on the value of an expression. The presence of two alternative flows infer that the workflow (see Fig. 1) assumes two possible states as output of the ‘GetBookStatus’ activity: the book is available or on loan. The procedure generates the following resulting state:

\[(\text{available}(\text{Book}) \text{ OR } \text{loan}(\text{Book}))\]

Every task produces an outcome, but sometime the workflow does not explicitly represent this information. When this happens, as for ‘CheckoutBook’ task, the procedure automatically generates a conventional state ‘task is completed’. The resulting state of ‘CheckoutBook’ is:

\[\text{checkoutbook\_done}(\text{Book},\text{Customer})\]

The complete set of goals for the book management workflow (Fig. 1) is the following:

\[
\begin{align*}
\text{(WHEN book\_request}(\text{Book}) \text{ INCOMES FROM Customer ROLE)} \\
\text{THE Customer ROLE, THE Clerks ROLE AND THE SYSTEM SHALL ADDRESS} \\
(\text{checkoutbook\_done}(\text{Book},\text{Customer}) \text{ OR cancelled\_request}(\text{Book},\text{Customer}))
\end{align*}
\]

\[
\begin{align*}
(\text{WHEN book\_request}(\text{Book}) \text{ INCOMES FROM Customer ROLE} \\
\text{OR AFTER 2 weeks SINCE WHEN hold\_book}(\text{Book}) \text{ INCOMES FROM Customer ROLE}) \\
\text{THE SYSTEM SHALL ADDRESS available}(\text{Book}) \text{ OR loan}(\text{Book})
\end{align*}
\]

\[
\begin{align*}
(\text{WHEN book\_request}(\text{Book}) \text{ INCOMES FROM Customer ROLE} \\
\text{AND WHEN available}(\text{Book}) \text{ } ) \\
\text{THE clerk ROLE SHALL ADDRESS checkoutbook\_done}(\text{Book},\text{Customer})
\end{align*}
\]

\[
\begin{align*}
(\text{WHEN book\_request}(\text{Book}) \text{ INCOMES FROM Customer ROLE} \\
\text{AND WHEN loan}(\text{Book}) \text{ } ) \\
\text{THE SYSTEM SHALL ADDRESS} \\
(\text{MESSAGE loan}(\text{Book}) \text{ IS SENT TO Customer ROLE})
\end{align*}
\]

\[
\begin{align*}
(\text{WHEN decline\_hold}(\text{Book}) \text{ INCOMES FROM Customer ROLE} \\
\text{OR AFTER 1 week SINCE MESSAGE loan}(\text{Book}) \text{ IS SENT TO Customer ROLE} ) \\
\text{THE SYSTEM SHALL ADDRESS cancel\_request\_done}(\text{Book},\text{Customer})
\end{align*}
\]

As this example shows, we are able to automatically extract goals from a BPMN workflow. These goals are expressed in such a way that they can be injected into an agent system and interpreted.

5 Discussion and Conclusions

The language GoalSPEC has been developed as a consequence of the requirements set described in Section 2, necessary to implement the adaptive system we imagined. None of the several languages in the literature for the description and specification of goals meets all those requirements.

GoalSPEC supports Adaptivity. The language is intended to be used within the life-cycle of a business process from creation to maintenance. The scenario starts when business analysts generate a preliminary version of business process by employing a BPMN visual tool. The tool generates a xml file that adopts the standard schema defined by OMG. The BPMN2Goal component receives this file and is able to automatically generates a set of GoalSPEC social and system goals. GoalSPEC is created in the
context of adaptive workflow and it completely covers the whole BPMN expressivity (REQ. 6), hence whatever process is defined with BPMN, it’s business goals (functional and non-functional) can be modelled with GoalSPEC (REQ. 1). Before being executed, system goals are proposed to analysts in order to be revised. Since GoalSPEC is based on the natural language, and it is specifically been conceived to attractive for a business audience (REQ. 4), analysts can easily understand and modify the results. This step is useful since analysts may include other business goals that they miss in the BPMN specification.

Several time in this paper we have mentioned that social and system goals are injected into the system at run-time. Indeed, the agent system is already running when business analysts work. Agents are specialized workers (each with their specific skills) waiting for something to do. When goals are released, agents perceive them into their environment. They are also able to interpret GoalSPEC (REQ. 3) and to absorb goals into their knowledge base. Even if the grammar is context-free, goal specification by GoalSPEC is not rigid for two reasons. Firstly a goal does not specify how to operate but it rather defines the expected results in ontological terms (REQ. 2), that is the final state of the world that is desired. In addition, some elements of the behavior specifications may be relaxed by using fuzzy modifiers (REQ. 5). In practice, agents potentially can plan and propose more alternatives for addressing a goals. Social interaction and individual capability of planning are out the scope of this paper.

**GoalSPEC supports Evolution.** Agents commit to the achievement of goals as long as they are perceived into the environment. Certainly current business process will change in the future, maybe as a cause of new business goals, new laws and so on. In a traditional approach, analysts would revision their BPMN models in accordance to changes. Any revision includes to check possible inter-dependencies among related (sub)processes with a consequent hard work of ensure coherence. The GoalSPEC approach is that the system intelligence will support this activity. Agents ability to reason on the injected goals may also highlight possible incoherence or conflicts between them. Warning of conflicts are useful for the analysts to improve the process. The workflow system will be always running, but the consequence of a goals revision is that agents will reorganize their behavior to address the new ones. Probably programmers will also introduce new agents into the system to cover the need for new skills.

**Considerations on the Expressiveness.** GoalSPEC adopts an ontological description of the business process. Logic predicates play a central role for the decomposition of the domain in a set of possible states of the world. Comparing GoalSPEC to Tropos, it appears that the second proposes a definitively richer semantics for the relationships between goals. GoalSPEC does not includes operators for and/or decomposition, contributions and means/end. This choice has been deliberately done since the will to make agents automatically discover these relationships. Any Tropos relationship adds a constraint for the system working. Otherwise, system intelligence must search for alternative solutions that were not designed by analysts. Comparing GoalSPEC to KAOS, it appears that the second uses a temporal logic, definitively more expressive than first order logic. Temporal propositions, in fact, contain some references to time conditions that GoalSPEC does not support. For example, we can’t specify that in the time between the event $E_1$ and the event $E_2$ the action $A$ can be executed at most twice. We accept
this limitation because our language needs to be more human oriented and feasible for complex systems. Indeed, systems based on temporal logic are difficultly scalable up and require formal verification.

But, in order to further increase the goal expressiveness GoalSpec also supports some fuzzy modifiers that may introduce uncertainty with the aim to increase the agent degree of freedom in pursuing their goals. Another consideration is that GoalSpec is able to cover not only the high expressiveness of the BPMN notation, but it is able to model more complex situations, for instance, those of a GANTT diagram.

Considerations on the Generality. The proposed language, although developed for a specific project, can be also used in more general information systems. We assert this because, it owns features that make it reusable in the general context in which workflows have to be managed. As it is well known, any information system embeds some kind of workflow even if sometimes that is not explicitly specified.

Final Remarks. GoalSpec wants to be a step toward the definition of an agent framework able to implement an adaptive workflow enactor in which goals may evolve because the user requirements are changed. Self-awareness is another important issue we are addressing. We are working to realize a kind of agent that is able to decide its own behavior with respect to evolving goals.

6 Acknowledgements

This work has been partially funded by the Innovative Document Sharing (IDS) Project funded by the Autonomous Region of Sicily (PO FESR Sicilia 2007-2013).

References


80
Benchmarking Communication in Actor- and Agent-Based Languages

Rafael C. Cardoso¹, Jomi F. Hübner², and Rafael H. Bordini¹

¹ FACIN–PUCRS
Porto Alegre - RS, Brazil
rafael.caue@acad.pucrs.br
r.bordini@pucrs.br

² DAS–UFSC
Florianópolis - SC, Brazil
jomi@das.ufsc.br

Abstract. This paper presents results on a communication benchmark between one agent-oriented programming language and two actor-oriented programming languages. It is based on an existing benchmark for programming languages and two variations on that benchmark. We selected Erlang and Scala to represent actor languages, and Jason as the agent language representative. We also present those three scenarios and the respective results in regards to time, memory, and core usage. Even though BDI engines typically used for agent languages provide sophisticated programming abstractions that require significant platform overhead to facilitate the development of complex agents, our initial results show that Jason has reasonable performance for this type of benchmark where actor-based languages are expected to do better than agent languages.

Keywords: benchmarking, agents, actors, Jason, Erlang, Scala

1 Introduction

Jason is one of the best-known multi-agent systems platforms based on agent-oriented programming. Various authors have included Jason in their comparisons and analyses of agent programming languages. For example, Jason was included in a qualitative comparison of features available in Erlang, Jason, and Java [17]; in a universal criteria catalog for agent development artifacts [8]; in a quantitative analysis of 2APL, GOAL, and Jason regarding their similarity and the measured time it takes them to reach specific states [5]; a performance evaluation of Jason when used for distributed crowd simulations [13]; an approach to query caching and a performance analysis of its usage in Jason, 2APL and GOAL [2]; and finally an implementation of Jason in Erlang and a benchmark for evaluating its performance [12]. In those cases where performance was considered, Jason typically showed excellent results.

However, there is no quantitative analysis — to the best of our knowledge — of how well agent languages can do against actor languages. Because the
actor approach is by design lighter than agents and because actor languages have been improved over a much longer period than modern agent programming languages, comparing performance on traditional programming language benchmarks is a much harder challenge for Jason than those it previously faced, and this is precisely what we do in this paper.

Our motivation for this recent line of work came from the idea of doing some benchmarking experiments in order to investigate whether some variations of usual benchmarking scenarios, taking into consideration features of agent programming, would allow us to conclude whether certain scenarios could be more appropriate for actors rather than agents and vice-versa, both in terms of naturalness of the paradigm for developing the scenarios and in terms of actual performance for the natural solutions in both paradigms. Even if it turns out that we cannot immediately find a scenario that is intrinsically more appropriate for agents, we could still use the outcome of our work to point out some flaws or deficiencies in current agent-based languages, and learn some of the advantages of actor-based languages, thus making it possible to improve performance for agent-based languages on the future.

We started by taking Erlang and Scala programs for a token passing problem available in the Computer Language Benchmarks Game website (http://shootout.alioth.debian.org/) and we wrote a Jason version for it. We then changed that benchmark to a different scenario where the only difference is that a number of tokens were being passed simultaneously; all three programs were changed accordingly. While Jason had the worst performance in regards to elapsed time in the first scenario, it did better than Scala in the second scenario, at least for our current experimental settings. Erlang showed the best performance in all cases, which was expected as it is developed by the Ericsson company and is aimed mainly to industrial development. Finally, for the third scenario, we added the notion of token types in order to test the reactivity of each platform.

We also considered including JACK [9] as a second agent language representative, but we decided not to do all of the experiments with JACK because of two main reasons: first, unlike the other selected languages, JACK is commercial software; and second, the results for JACK in the first two scenarios showed that it seems not to take advantage of multiple cores, so it cannot be compared to the other 3 languages used for these experiments. A summary of these results can be examined at the end of Sections 3.1 and 3.2.

The results reported in this paper were obtained from runs on a dedicated computer with four physical cores (no hyperthreading). We also show the results for the same scenarios but limiting the number of cores to two, with the purpose of analysing the difference that it makes to run the same experiment on increasing numbers of cores. The experiments presented in this paper are not suposed to stress-test the languages but rather to compare them in normal day-to-day usage measuring the performance, scalability, and reactivity of the communicat-

3 The codes for every scenario used in this paper are available at https://github.com/rafaelcaue/Actor-Agent-based-benchmark-for-communication.
tion aspect of the languages. Because this comparison cannot be done directly, as the actor model is by design lighter than the agent model and each language has a different runtime environment, we make use of scale factors to compare these languages.

In summary, this paper aims to present results about the performance of agent-based languages, particularly Jason, against actor-based languages, using variations of a well-known benchmark for communication in programming languages, and stimulate the agent community to further benchmark different aspects of agent-oriented programming languages.

The remainder of this paper is organised as follows. The next section gives a brief introduction to the three programming languages we compare in this paper. Section 3 shows the description of each scenario used in the experiments as well as all the results obtained. Section 4 includes the analysis of the results we obtained and we conclude the paper in Section 5.

2 Agent and Actor Programming Languages

In this section, we briefly discuss the main aspects of the three programming languages that we chose for this comparison work. We only present some of the fundamentals of each language, which can help understand the scenarios implemented in the experiments. However, we assume some familiarity of the reader with the actor and agent paradigms and, of course, prior knowledge of those three programming languages (as well as functional programming and Java) would be helpful.

Actor-oriented programming languages are based on the actor model [1], with an actor being a lightweight process that does not share state with other actors and communicates by asynchronous message passing through mailboxes. Agent-oriented programming languages are based on the agent model that is an extension of the actor model. While both agents and actors are lightweight processes and reactive, agents are more complex entities and typically capable of “practical reasoning” (i.e., logic-based reasoning about the best action to take at the given circumstances) [6,7].

2.1 Jason

Jason is a platform for the development of multi-agent systems that is based on an agent-oriented programming language. The logic-based BDI-inspired language AgentSpeak, initially conceived by Rao [20], was later much extended in a series of publications by Bordini, Hübner, and colleagues, so as to make it suitable as a practical agent programming language. These extensions led to the variant of AgentSpeak that is made available in Jason [7].

In Jason, an agent is an entity composed of a set of beliefs, representing agent’s current state and knowledge about the environment in which it is situated, a set of goals, which correspond to tasks the agent has to perform/achieve,
a set of intentions, which are tasks the agent is committed to achieve, and a set of plans which are courses of actions triggered by events.

Events can be related to changes in either the agent’s belief base or its goals. The agent reacts to the events creating new intentions, provided there is an applicable plan for that event. Therefore, each intention represents a particular “focus of attention” for the various tasks currently being done by the agent: they all compete for the agent’s choice of intention to be further executed in a given execution step. Last, we must mention the “pool of threads” functionality of Jason, declared by using \((\text{pool}, x)\) next to the infrastructure of choice. Enabling a pool of threads means that rather than creating a thread for each agent, Jason creates only a fixed number of threads that agents compete for (unless they have nothing to do). In our experiments, we chose \(x\) to be the number of cores used in order to increase performance on multi-core processors (\(x\) was either 2 or 4, depending on the experiment settings).

### 2.2 Erlang

Erlang [4], acronym for Ericsson Language (where it was developed), is a functional language with dynamic typing. Erlang is supported by an extensive library collection known as OTP, originally an acronym for Open Telecom Platform (before 2000 when Erlang became open source). The Erlang Run-Time System (ERTS) application is responsible for low-level operations, including the Erlang virtual machine called Bodgan’s Erlang Abstract Machine (BEAM) [3].

Concurrent programming is the focus of Erlang, using processes (the concurrency model usually referred by Erlang users is the process model, but it corresponds directly to the actor model) that are as much lightweight as possible. It even has its own scheduler in the virtual machine, so a process in Erlang has nothing to do with heavyweight operating system processes.

Communication between processes is based on message passing, with each process having its own mailbox that is used to store the messages. Messages can be sent asynchronously and if a message matching the pattern is found in the queue, it is processed and variables instantiated before the expressions in the body are evaluated. Functions are also defined by pattern matching and expressions as usual in functional languages. For further details, we refer the interested reader to [18,10].

### 2.3 Scala

Scala is considered a multi-paradigm language, as it combines features of object-oriented and functional programming languages. It differs from Erlang on its type system, as Scala is a statically typed language, and aimed as a general purpose programming language. The name Scala comes from “scalable language”, as it was designed to help the development of systems that need to scale up. Scala programs run on a Java Virtual Machine, so it has direct integration with Java, allowing the use of existing Java code within Scala systems [19].
This facilitates extensibility of the language, and resulted in the creation of many libraries, such as Scala Actors: a library providing concurrent programming based on actors for Scala programming. An actor can communicate asynchronously with other actors by exchanging messages through the actor’s mailbox; an actor than generates an appropriate response for each message it receives [14].

It is important to note that the Scala actors used in the scenarios of this paper are event-based actors, as the expression used is react, while for thread-based actors the expression would be receive [15]. Although Scala has many other interesting concepts, only the basics sufficient for understanding the scenarios in Section 3 are covered here. For a more in-depth reading of Scala, we suggest [21] and, more specifically for Scala Actors, [16].

3 Experimental Results

The Computer Language Benchmarks Game (http://shootout.alioth.debian.org/) provides performance evaluation for approximately twenty four languages on some benchmark problems. Although they evaluate the performance on computers with multiple cores, the tasks and most of the languages are not appropriate for concurrent programming. A python script is available on their website that does repeated measurements of CPU time, elapsed time, resident memory usage, CPU load for each core, and it does so for various programs written in different programming languages. The script then summarises those measurements on a sheet for easy viewing.

Each program is run as a child-process of a Python script using Popen. The script is fully customisable and it is easy to add new languages, so we were able to adapt it to our experiments. For the experiments described below, we chose to take two measurements with the script: CPU load for each core and elapsed time. The script measures the percentage load of a core through the GTop library, on Unix systems, taking the CPU-idle and CPU-total values before forking the child-process and after it exits, where the percentage represents the time that a core was not-idle. Elapsed time uses time.time() to get the time before forking and after exiting. We encountered a problem with the memory use reported by the script; for various reasons the numbers reported were clearly inaccurate for the Java-based languages. Therefore, we used the Java profiler jvisualvm and took the highest point of memory used for these two languages. Therefore, we used the Java profiler jvisualvm and took the highest point of memory used for these two languages.

The scenarios described in the next sections focus on the message passing aspect of communication, testing the support for asynchronous message passing and concurrency of each language; both of these features are essential for actor- and agent-based languages. In order to run the experiments\(^4\), we used an Intel®Core™i5-2400 CPU @ 3.10GHz (4 physical cores, no HyperThreading)

\(^4\) Additional system specification: 500GB SATA II hard disk drive, the kernel version was 3.2.0-29-generic, we used Java OpenJDK 64-Bit Server VM (build 23.2-b09), Java 1.7.0_07, Apache Ant 1.8.2, and finally for the python script, we used Python 2.7.3.
machine with 4GB of DDR3 1333 MHz RAM, running the operating system Ubuntu 12.04.1 LTS 64 bits; the versions of the languages used were Jason 1.3.9, Erlang R14B04 erts 5.8.5, and Scala 2.9.1.

For the experiments shown below, in particular those for varying numbers of token passes \(N\), we changed the default arguments of the Java Virtual Machine in respect to Xms (Initial Heap Size) from 64M to 2G and in Xmx (Max Heap Size) from 1G to 2G. We chose the same values for Xms and Xmx because it is commonly known that if the heap starts at the max value, then it will have to run the garbage collector less often, hence improving performance. Varying \(N\) turned out not to affect significantly the memory used by any of the three languages, so in order to show relevant memory results we varied the number of workers. When numbers of workers (i.e., the token passing entities) are being varied, we decided to use the default values for Xms and Xmx, as changing them proved to have an adverse impact in memory usage. We also measured time and core load for this variation, but omitted them in this paper since they did not show anything different from the other experiments.

3.1 Scenario 1

The first scenario is a simple case of passing a token \(N\) times through a ring of “workers” (i.e., agents, processes, or actors, depending on the language). The Erlang and Scala code for this scenario are those available at (http://shootout.alioth.debian.org/), except that we removed unnecessary print statements, and changed the number of workers to 500, simply because it represents an intermediate value between the number of tokens used in Scenario 2 and the workers needed for the memory experiments. Each program in this scenario should:

- create 500 linked workers (named 1 to 500);
- worker 500 should be linked to worker 1, forming an unbroken ring;
- pass a token to worker 1;
- each worker passes the token to its neighbouring worker;
- the program halts when the token has been passed (between any two workers) \(N\) times.

We ran experiments for Scenario 1 with six different configurations for \(N\), measuring elapsed time and core load: \(N = 500; N = 5,000; N = 50,000; N = 500,000; N = 5,000,000; N = 50,000,000.\)

And again with three different configurations of number of workers \((W)\), this time with \(N\) fixed at 5,000,000, for measurements of resident memory: \(W = 50; W = 500; W = 5,000.\)

The results for the first benchmark (Scenario 1), using the Python “bencher” script, can be seen in the following graphs\(^5\). Figure 1 presents the measurements of elapsed running time in seconds based on the values in Table 1. Figure 2

\(^5\) All the graphs presented in this paper are in logarithmic scale, given that \(N\) grows exponentially.
shows the CPU load measurements for each of the four cores, for the three languages we are comparing; Jason is using on average 50% of each core in all three configurations; Erlang is using mostly one core, 70% for $N = 500k$, 95% for $N = 5m$ and 100% for $N = 50m$; Scala is using on average 40% of each core for the first two configurations, and 30% of each core for the last configuration.

Table 1. Elapsed time in seconds for Scenario 1, varying $N$.

<table>
<thead>
<tr>
<th></th>
<th>500</th>
<th>5k</th>
<th>50k</th>
<th>500k</th>
<th>5m</th>
<th>50m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason</td>
<td>1.736</td>
<td>2.183</td>
<td>2.962</td>
<td>8.104</td>
<td>59.744</td>
<td>571.656</td>
</tr>
<tr>
<td>Erlang</td>
<td>0.074</td>
<td>0.071</td>
<td>0.09</td>
<td>0.224</td>
<td>1.502</td>
<td>13.826</td>
</tr>
<tr>
<td>Scala</td>
<td>0.736</td>
<td>1.181</td>
<td>1.498</td>
<td>3.353</td>
<td>23.184</td>
<td>211.36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>500</th>
<th>5k</th>
<th>50k</th>
<th>500k</th>
<th>5m</th>
<th>50m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason</td>
<td>1.841</td>
<td>2.213</td>
<td>2.794</td>
<td>5.687</td>
<td>31.016</td>
<td>263.35</td>
</tr>
<tr>
<td>Erlang</td>
<td>0.058</td>
<td>0.059</td>
<td>0.073</td>
<td>0.214</td>
<td>1.489</td>
<td>14.395</td>
</tr>
<tr>
<td>Scala</td>
<td>0.764</td>
<td>1.188</td>
<td>1.472</td>
<td>2.671</td>
<td>12.991</td>
<td>104.656</td>
</tr>
</tbody>
</table>

For readability we present Figure 9 in Section 3.2 next to the memory results for Scenario 2. The results show the measurements of the highest peak in resident memory used in megabytes: when there were 50 workers, Jason used 56MB at the highest peak of memory usage, Erlang 8MB, and Scala 110MB; for 500 workers, Jason used 200MB, Erlang 9MB, and Scala 94MB; and for 5,000 workers, Jason used 349MB, Erlang 20MB, and Scala 201MB. All the numbers shown are based on the results collected through 5 repeated measurements of each program with each of the six configurations; in particular, the numbers shown below represent the turn with lowest (best) value of elapsed time among the 5 different runs. We also show the graphs for the same configurations running with only two cores: Figure 3, 4. In regards to CPU load using two cores: Jason used on average
75% of both cores; Erlang again used mostly only one core, 80% for the first configuration and 100% for the last two; Scala averaged 55%.

As reported in Section 1 we also considered using JACK for the experiments, as in this first scenario it presented a promising value of 6 seconds for the measurement of elapsed time with $N = 500,000$, which was 2.1 seconds faster than Jason. JACK used, on average, 45% of each core in this experiment.

3.2 Scenario 2

This scenario is a small variation of Scenario 1, where we added more tokens and allowed them to be passed concurrently. So rather than passing only one
token, in Scenario 2 at the start of a run 50 tokens are distributed around the ring using the following equation:

\[ I \times \left( \frac{W}{T} \right) \]

where \( I \) is the number of the current token to be sent, \( W \) is the total number of workers, and \( T \) is the total number of tokens. Each of these 50 tokens have to be passed \( N \) times, and because neither agents nor actors share state, a counter is needed for counting the tokens that have finished: this is necessary because in order for the Python bencher script to keep running it needs the programs to halt, which can only happen when all 50 tokens have been passed \( N \) times each.

The results for the second benchmark are shown in the following graphs. Figure 5 shows the measurements of elapsed running time in seconds based on the values in Table 2. Figure 6 shows the measurements for the four cores, for each of the languages; Jason averaged 98%, Erlang averaged 97%, and Scala averaged 30%. Figure 10 shows the measurements of the highest peak in resident memory used in megabytes; when there were 50 workers, Jason used 167MB at the highest peak, Erlang 8MB and Scala 266MB; for 500 workers, Jason used 239MB, Erlang 9MB and Scala 59MB; and for 5,000 workers, Jason used 509MB, Erlang 20MB and Scala 108MB. As before, we pick the run with the lowest elapsed time and show the graphs for two cores in Figure 7 and 8. In regards to CPU load using two cores, Jason and Erlang averaged 99%, and Scala averaged 55%.

Although JACK presented promising results in Scenario 1, it could not effectively handle the concurrency in Scenario 2, with a value of 217 seconds for the measurement of elapsed time with \( N = 500,000 \), which was far off from the results of any of the other three languages. JACK used, on average, 60% of each core in this experiment. This disparity between the results for the two scenarios were a main factor for not including JACK in the rest of the experiments.
Table 2.Elapsed time in seconds for Scenario 2, varying $N$.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>500</th>
<th>5k</th>
<th>50k</th>
<th>500k</th>
<th>5m</th>
<th>50m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason</td>
<td>2.217</td>
<td>2.628</td>
<td>5.653</td>
<td>34.932</td>
<td>331.544</td>
<td>3296.662</td>
</tr>
<tr>
<td>Erlang</td>
<td>0.075</td>
<td>0.088</td>
<td>0.319</td>
<td>2.583</td>
<td>25.081</td>
<td>251.808</td>
</tr>
<tr>
<td>Scala</td>
<td>1.05</td>
<td>1.758</td>
<td>6.471</td>
<td>70.397</td>
<td>580.757</td>
<td>6066.445</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Configuration</th>
<th>500</th>
<th>5k</th>
<th>50k</th>
<th>500k</th>
<th>5m</th>
<th>50m</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason</td>
<td>2.623</td>
<td>3.371</td>
<td>9.315</td>
<td>66.924</td>
<td>651.501</td>
<td>6415.058</td>
</tr>
<tr>
<td>Erlang</td>
<td>0.068</td>
<td>0.093</td>
<td>0.393</td>
<td>3.545</td>
<td>34.359</td>
<td>346.454</td>
</tr>
<tr>
<td>Scala</td>
<td>1.267</td>
<td>1.729</td>
<td>6.219</td>
<td>57.977</td>
<td>468.888</td>
<td>5490.613</td>
</tr>
</tbody>
</table>

Fig. 5. Elapsed time in seconds for Scenario 2.

3.3 Scenario 3

As a follow up to Scenario 2, this time $N$ is fixed at 500 and a new type of token was introduced in the ring; we call it “token type 1”. The 50 previous tokens from Scenario 2 are now referred to as “token type 2” and they work exactly the same way as in the previous scenario. We created 1,000 type 1 tokens, two per worker, to simulate the “mundane tasks” that the workers would normally be doing while waiting for the “special” type 2 tokens to arrive. When a worker acquires a token type 1 it starts a loop with an abstract computing load (we used an empty loop of 1000 iterations), and when it finishes that work load on the type 1 token, it passes on the token to the next worker in the ring, but the token counter is not decreased so that tokens of type 1 never stop moving around the ring. On the other hand, if a worker acquired a token type 2, as soon as it realises it, the worker would pass that token directly to the next worker in the ring. A run of Scenario 3 ends when all 50 tokens type 2 have been passed 500 times each.

These new modifications were introduced mainly to test the reaction time when receiving a token type 2 while still busy with the tokens of type 1 (i.e. test the reactivity in communication for each platform), so we decided to take
the fastest configuration of Scenario 2, $N = 500$, which could then be compared with the new results obtained for Scenario 3. In Jason, this reactivity comes naturally: when one token of any type arrives, it generates a new intention that is executed concurrently with the other intentions of that same agent. That is not the case with Erlang and Scala, as they need to create a “subactor” for each token that arrives, except for tokens type 2 which do not require additional work before passing. If instead the actor did not create subactors, it would be blocked working on all of the tokens type 1 it previously received before getting the message that a token type 2 arrived. As before, we pick the run with the lowest elapsed time from 5 separate runs.

6 Of course this is not the only way to implement such reactivity in the actor languages, but it would be the most “natural” and easiest to program in that paradigm.

Fig. 6. Core load for Scenario 2.

Fig. 7. Elapsed time in seconds for Scenario 2 (two cores).
Fig. 8. Core load for Scenario 2 (two cores).

Fig. 9. Highest memory peak in Scenario 1.

Fig. 10. Highest memory peak in Scenario 2.

It is important to note that each of the three languages used a different mechanism to simulate the work load of tokens type 1. In Jason the loop was made through goal recursion, in Erlang through tail recursive functions, and in Scala with a simple for loop from Java. Because a worker will always be working on at least one token type 1, these variations of the implementation of an abstract work load do not interfere with our results because we will not compare the values of different languages, we will compare them with the respective results from Scenario 2. The results of elapsed time for Scenario 3 are presented in Table 3, with N = 500 for using both four and two cores. We do not present in this paper the graphs for core load and memory usage for this scenario because they produced similar results to those already observed in Scenario 2.

Table 3. Elapsed time in seconds for Scenario 3.

<table>
<thead>
<tr>
<th></th>
<th>4 cores</th>
<th>2 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jason</td>
<td>12.246</td>
<td>19.922</td>
</tr>
<tr>
<td>Erlang</td>
<td>0.482</td>
<td>0.748</td>
</tr>
<tr>
<td>Scala</td>
<td>3.678</td>
<td>3.566</td>
</tr>
</tbody>
</table>
4 Analysis of the Results

The elapsed time graph for Scenario 1 (Figure 1) shows Jason and Scala close to each other, while Erlang remains distant as the fastest language, but in Scenario 2 (Figure 5), Jason exchange places with Scala and gets closer to Erlang. When limiting the number of cores to two, Scenario 1 is drastically improved for Jason and Scala as their elapsed times lower considerably with each configuration, which could be explained by the time spent switching threads to cores, but not so with Erlang that was already using mostly only one core, even though it still improved by a low margin. With two cores in Scenario 2, results indicate that Jason and Erlang have a decline in performance, which makes sense since Scenario 2 is supposed to be concurrent and having fewer processors should increase the elapsed time, although that is not the case with Scala as it improves its performance, possibly due to the fact that it was not maximising the use of cores.

Those behaviours become clearer when observing the scale factors values in Table 4 for Scenarios 1 and 2, scale factors represent the proportional increase in time when scaling up the experiment configurations such as number of token passes, and denotes the degradation of performance. For example Scala scale factor 9.12 for Scenario 1 in configuration 5m to 50m is calculated by dividing its elapsed time in 50m, 211.36 seconds, for its elapsed time in 5m, 23.184. We selected only the three highest values for N due to the fact that the other configurations only showed small variations between them.

The scale factors for Scenario 1 show Scala as the one with the lowest difference value between scale factors: 2.21 (i.e. the scale factor from configuration 5m to 50m, 9.12, minus the scale factor from 500k to 5m, 6.91) compared to 2.5 for Erlang and 2.38 for Jason. Those differences remain when limiting the number of cores to two, but when looking at the values for Scenario 2, Erlang takes the lead with a difference value of 0.33 compared to 0.45 of Jason and 2.2 of Scala. Regardless of whether the scenario is concurrent or not, Jason manages to match closely the performance of the actor languages, and even surpasses them by a large margin as shown by the scale factors in Table 5 (moving from a non-concurrent to a concurrent scenario). The scale factors for Scenario 3 are presented in Table 6 and show Scala as the most reactive language, followed by Jason and then Erlang.

The results in Figure 2 show that Jason and Scala have an even distribution of core load, while Erlang uses for the most part only one core, but we must consider that Scenario 1 does not require concurrency, so it is acceptable that only one core should be the most used. Moving to the CPU-load graph for Scenario 2 (Figure 6), we can see that Erlang starts to use all of the 4 cores evenly, with both Erlang and Jason using a lot more of core load than in Scenario 1, but Scala remaining on the 30% mark, which explains some of its poor performance regarding elapsed time in Scenario 2. This could be caused by either an inadequate underlying runtime management of multi-core usage or a missing configuration parameter, that as far as we know does not exist.
Where memory was considered, Jason and Erlang had the expected increase in used memory with the increase in the number of workers. Jason has the highest memory usage of the three languages, which was predictable since each agent has a more complex internal structure than an actor. Surprisingly, Erlang did not show any difference regarding memory usage from Scenario 1 to Scenario 2, and while Scala showed the expected results for Scenario 1, when observing Scenario 2 the memory usage was all over the place, again probably because of the low core usage.

Having its own virtual machine and runtime environment execution certainly helps Erlang to achieve the performance presented in this paper, although we cannot say to what extent this may affect its overall performance. Clearly there are advantages in using Java and the JVM for Jason and Scala, but this places a limit on the performance that they can achieve.

On a more qualitative note, it is interesting to observe the code size of the solutions. In all scenarios, Jason uses significantly fewer lines of code compared to Erlang and Scala, and – although this is subjective – the programs seem more intuitive, simpler, and readable. The interested readers can see all programs used for the three languages at https://github.com/rafaelcaue/Actor-Agent-based-benchmark-for-communication.
5 Conclusion

In this paper, we presented an analysis of the results from three different Scenarios. Scenario 1, where a token is passed sequentially, Scenario 2 where multiple tokens are passed concurrently, and Scenario 3, where reactivity is needed for tokens type 2 to be passed faster. Where scaling was considered, Scala does justice to its name and did better in almost all cases where the scale factors were considered, while Erlang stood distant as the one with significantly better performance than the other two languages at both elapsed time and memory used for the current scenarios. Jason, as a representative from a “heavier” paradigm, did not disappoint, following closely on both aspects, scalability and performance, and even excelling at some aspects in comparison to the two actor languages, proving that agent-oriented programming languages can perform surprisingly close to its predecessors as far as communication is concerned.

Future work includes running the experiments reported here on machines with a higher number of cores, and analyzing new aspects such as giving priority to some of the tasks acquired by communication. Furthermore, we intend to benchmark also other agent and actor programming languages, including eJason [12] and ERESYE [11] – ERlang Expert SYstem Engine, which is a library to develop expert systems that allows rules to be written as Erlang function clauses. To complement the work on benchmarking, we aim to consider the fundamentals of programming languages for a more qualitative comparison of the languages.

There has been very little research on benchmarking for agent programming languages, so we expect to report various other results and analyses in the near future, and we also expect to see similar efforts for the great variety of agent programming languages. In order to support such effort we have developed a website — http://www.inf.pucrs.br/maop.benchmarking/ — to serve as a repository of benchmarks specifically designed for comparison of the existing agent programming languages. Benchmarking agent programming languages can sometimes lead to performance improvements, and is an important step towards mainstreaming of the agent model.

Acknowledgments. We are grateful for the support given by CNPq (grant numbers 307924/2009-2 and 307350/2009-6) and by CAPES.

References

A Communication Infrastructure Based on Artifacts for the JaCaMo Platform

Thiago Fredes Rodrigues\textsuperscript{1}, Antônio Carlos da Rocha Costa\textsuperscript{12}, and Graçaliz Pereira Dimuro\textsuperscript{12}

\textsuperscript{1} Centro de Ciências Computacionais
\textsuperscript{2} Programa de Pós-Graduação em Computação
Universidade Federal do Rio Grande, Brazil
{trodrigues02, ac.rocha.costa, gracaliz}@gmail.com

Abstract. When developing a Multi-Agent System (MAS) some components are usually considered, namely, the agent population, the system social organization, its environment and the interactions among organizational roles and agents playing those roles. The JaCaMo platform provides facilities for a modular development of the first three MAS components, but not for interactions. This paper presents a communications infrastructure for the JaCaMo platform, aiming to provide a modular way for describing the interactions allowed in the MAS. The infrastructure’s architecture is based on the CArtAgO framework, a component of the JaCaMo platform.

1 Introduction

A Multi-Agent System (MAS) is a set of computational intelligent, autonomous, capable of communicating, pro-active and situated entities, perceiving its environment through sensors and acting on it using actuators [1–3]. One problem that arises when developing a MAS is the coordination of agents’ actions. Then, organizational aspects and multi-agent communication are usually considered in order to allow the agents’ coordination. The organizational modeling can limit agents’ autonomy and concentrate their actions around the desired global behaviour of the MAS organization. On the other hand, agent communication is a way of getting information to synchronize actions and avoid conflicts. Also, the interactions between agents, through agents’ communication, play an important role in the MAS-based simulation of social (real world) organizations.

Moreover, a MAS is a multi-dimensional system that involves the development of four components: the agents’ population (where the system’s decision-making processes happen), the system’s organization (which describes the allowed actions in the MAS), the system’s environment (possibly acting as an interface with a physical world) and the set of interactions executed among agents playing organizational roles (communication protocols for reaching agreements).

A development platform that currently offers high-level and modular facilities to program the first three components is JaCaMo. [4]
JaCaMo is actually the synergic usage of three separate technologies: the Jason interpreter, the CArtAgO framework and the MOISE+ model [5]. The first is used to program the agents’ population, the second allows the environment’s programming and the last is an organizational model, used to create the MAS’s organization. All those MAS components are programmed/developed in a modular way and in a high-level of abstraction, making JaCaMo a rich and easy to use MAS development platform. However, in JaCaMo there is not a native and modular way of defining the interactions allowed among agents. Neither in the agent population nor in the organization dimension. A tool that helps this task and provides a set of services to communication is JADE [6]. However it is an external resource and not a component of JaCaMo.

In this paper, a Communications Infrastructure for JaCaMo is introduced, through the CArtAgO framework [7] (a component of JaCaMo), aiming to define interactions among agents in a modular way (currently without any links to the organization.) Nevertheless, other related works are discussed, as well as some future works regarding the definition of protocols in the organization dimension.

This paper is organized in the following way. Section 2 briefly presents the main aspects of agent communication. In Sect. 3, the JaCaMo platform is presented. In Sect. 4, the proposed Communications Infrastructure is introduced, with some contextualized examples regarding a social simulation scenario. Finally, Sect. 5 is the Conclusion, where some future works are also outlined.

2 Agent Communication for Agent Coordination and Interaction

Sometimes, although having the same goal, the actions of one single agent can compromise the action of another agent. For instance, [3] presents a situation in which two agents need to cross a passage that only allows one agent at a time. If both try to cross at the same time, there will be collision and one will compromise the performance of the other.

One common solution to prevent conflicts is to model the MAS organization structure, in which roles and their respective goals are defined. Then an agent can adopt a role that allows it to use only some of its individual capabilities, limiting his freedom. This limitation is a way to coordinate the agents’ actions, stimulating the desired interactions in the system and helping it maintain and achieve its goals [8].

Another solution to reach coordination is through communication. The agents exchange information to synchronize their actions so conflicts do not compromise their performance. Also, it allows cooperation between agents when the achievement of a goal is only possible when the agents are working together. The messages can be written according to some Agent Communication Language (ACL), following the Speech Act theory [3].

The two most known ACLs are KQML and FIPA-ACL [3, 9].

---

3 A detailed comparison of both languages can be found in [9].
3 The JaCaMo Platform

JaCaMo [4] is a MAS development platform, comprising three technologies: Jason [10], CArtAgO [7] and MOISE+ [8,5]. Its components are orthogonal, each used to create a different part of the system. In the following subsections, each component is briefly explained.

3.1 Jason: an interpreter for the AgentSpeak-L language

Jason is an interpreter for the AgentSpeak-L language [11], also providing a MAS development platform, supporting communication between agents based on the speech act theory, with a KQML-based syntax. The AgentSpeak-L language allows the creation of MAS based on the BDI (Beliefs, Desires, Intentions) architecture.

An AgentSpeak-L agent is programmed as an initial set of beliefs and a set of plans, used to achieve goals and to respond to events and perceptions generated by the environment, as well as the receipt of messages sent by the other agents. It is possible to create two types of goals in AgentSpeak-L: achievement goals and test goals. The former express a state of affairs the agent wants to achieve, in which the predicate associated to the goal is true. For that end, plans can be triggered. The latter returns the unification of the test goal’s predicate with an agent belief, or fails if no unification is possible. In general, the recovered belief is used inside another plan.

Every time the belief base changes (by percepts or by beliefs manipulation inside plans) or a goal is added/removed, an event is generated to treat this alteration. Basically, the agent’s program execution occurs by triggering plans to treat events. Every time this happens, a search through the code occurs, looking for a plan that treats the event and, when a plan is selected, its body is executed. A plan’s body is composed by a sequence of actions, that can be external actions (executed in the environment), internal actions (actions executed within the agent, usually to manipulate data structures) as well as goal and belief addition or removal (belief manipulation is very similar to goal manipulation).

The external actions are executed in the environment and depending on the way the environment was implemented, those actions get conducted differently. Jason provides two ways to implement it: as a Java class or using the CArtAgO framework. In the former way, those actions are mapped to methods of the class. In the latter, actions are mapped to operations available on the artifacts composing the environment.

3.2 CArtAgO: a framework to create virtual environments

CArtAgo (Common ARTifact infrastructure for AGents Open environments) is an open source framework that provides an API to create virtual environments for multi-agent systems. The environment is considered as a computational layer encapsulating the functionalities and non-autonomous services the agents can
exploit during runtime. It is also possible to create distributed environments and multi-agent systems based on Java agents.

CArtAgO is based on the Agents & Artifacts (A & A) meta-model, to model multi-agent systems. This model introduces a high level metaphor, based on the idea that humans work together in a cooperative way with its environment: agents are computational entities that execute some type of goal oriented task (like human workers), and artifacts are some type of resources and tools dynamically constructed, handled and shared by agents to support and perform their activities. Therefore, it is possible to develop artifacts instantiated in the environment that provide services for agents, maybe performing some communication with external services like web-services and hardware. In [4], there are several projects mentioning the integration Jason-CArtAgO.

A CArtAgO environment is composed by workspaces, which contain artifacts, and every workspace is contained in a node. In every CArtAgO environment, there is a workspace, called default, which provides the framework’s basic functionalities. An agent can simultaneously work in various workspaces. Various CArtAgO nodes can run on a single computer and different nodes can communicate through network.

An artifact is a Java class extending the Artifact class. Every artifact class must implement a method called init, which functions as its constructor. Inside this method all the procedures regarding its initialization should be made (initializing variables, for instance).

Artifacts also have observable properties, which are visible to agents situated in the environment. Every time their value changes, agents receive a perception notifying this modification. An observable property is visible to all agents that execute the external action focus over the artifact that contains it. This operation informs the framework the agent has interest on that artifact and in the modifications that can occur in it. Analogous to observable properties, signals are presented to all agents that executed that operation but, differently, they can be shown in a directed way (that is, the signal is sent by a specific agent).

Artifact’s usage occurs by means of the operations available on them, which constitutes their interface with the agents. Those operations are methods of the class implementing the artifact. An operation execution is synchronous, that is, the agent who used the operation loses the execution flow of the plan invoking the operation to the framework and only recovers it after the operation ends.

Lastly, in CArtAgO an agent can link artifacts in such a way that an operation called in one triggers the execution of another operation in the other artifact. For that, it is necessary to define a communication port between artifacts. Moreover, not all operations defined in the artifact’s interface can be used by another artifact (this is defined in the artifact’s implementing class code). For agents, the semantics of this operation does not change, but for an artifact it means the operation can be used by it. Alternatively, an artifact can get connected to another one without an agent linking them. This is possible if the linked artifacts are created inside the linking artifacts.
3.3 MOISE+: an organizational model for MAS

The MOISE+ organizational model [5] allows the creation of multi-agent organizations and is composed by three dimensions: structural, functional and deontic.

In the structural dimension it is possible to define roles, which are restrictions an agent accept when it gets into a group [8]. This restrictions are made to limit the agent’s behavior, so its actions are more directed towards the goals designed for the MAS. Usually, there can be as many groups as desired, and inside each one there are a set of roles. Also, lower and upper bounds to the amount of agents and group instances that can exist at a time can be defined. A group is said well-formed when those restrictions are satisfied (e.g., there are five agents performing role A, when the lower bound is two and the upper one is six). The same role can exist in different groups.

Among roles, some links can exist, as presented in the following list:

1. compatibility: a role is compatible to another if an agent can perform both roles simultaneously;
2. authority: a role that has authority over another can give orders to agents performing it, having also communication and acquaintance relations;
3. communication: this link allows a role to communicate with another, having also an acquaintance relation with it;
4. acquaintance: this relation simply enables an agent to be aware of another role’s existence;
5. inheritance: this link is analogous to the inheritance relation existent in object-oriented languages, enabling a role to inherit the links and binds owned by another.

Also it is possible to restrict some relations based in a group scope, that is, if two roles are performed by agents in the same group, a link is valid, if they are in different groups, another link is valid. These are called intra-group and inter-group relationships.

In the functional dimension, schemes are defined, which consist in a structure similar to a tree whose nodes are goals. These are the ones the MAS was designed to achieve. In the scheme, a goal can be divided in smaller ones, and a order relationship among them can be defined. It can be of three types: sequence (when each goal must be achieved before the next), choice (when among a set of goals only one must be selected to be achieved) and parallelism (when all goals can be pursued simultaneously and there is no order among them). Besides goals, in this dimension, missions are also defined, which are simply a grouping of goals. When an agent gets committed to a mission, it gets bound to achieve the goals the mission contains.

Finally, the deontic dimension defines which role can bind to which mission as well as the relation between them. This relation can be of obligation (when a role is obliged to commit to a mission) or of permission (when a role can commit to a mission, but is not obliged to it). This dimension connects the two other dimensions defined in MOISE+. 
The integration of this model with the two other JaCaMo components occurs by means of two specific artifacts, provided by default in the CArtAgO framework: *GroupBoard* and *SchemeBoard*. The former artifact allows agents to adopt and leave roles defined in the structural dimension of MOISE+ as well as to add and remove schemes of it. Its observable properties tell the current state of groups defined in the organization (whether they are well-formed or not) and roles adopted by each agent. The latter artifact provides operations for agents to bind to missions and alter the state of goals, after achieving them (e.g., setting their state as *achieved*). Likewise, which agents are bound to which missions and the state of each goal are observable properties. To access those information, an agent needs to execute the focus operation.

3.4 Specifying Agents Interactions in JaCaMo

The JaCaMo platform provides resources to create complex multi-agent systems, offering a programming platform composed by orthogonal technologies, so alterations in one dimension cause small or no changes in the other ones. The technologies found in JaCaMo allow the development of MAS, since it allows the programming tasks with a high level of abstraction and in a modular way:

1. using the MOISE+ model, the main processes and agent types of the system can be defined;
2. in the Jason interpreter, agents are programmed in a high level of abstraction, through a logic-based language that allows a close mapping of concepts;
3. CArtAgO allows the creation of virtual environments, programmed in Java, making possible also the use of agents written in this same language.

However, as we mentioned before, there is no place in JaCaMo to describe, in a equally modular way, the interactions that can happen in the MAS. For example, consider that to achieve goal A, the role B must use protocol C. In the MOISE+ model, it is only possible to define that two roles can communicate, but not the protocol used in the communication nor the messages that should be exchanged. Moreover, the JaCaMo platform does not have a distributed infrastructure to exchange messages among its running agents. Currently, only Jason agents can communicate, limiting the platform’s possibilities of integration with other technologies.

4 The Proposed Communications Infrastructure

The proposed communication infrastructure uses artifacts to mediate communication and is based on the CArtAgO framework. It allows modular specification of the possible interactions in the MAS, encapsulating them in artifacts. Thus, it is possible to create new protocols and different ways of communication just creating the respective artifacts. The communication infrastructure offers the following resources:
1. Communication among agents programmed in both AgentSpeak-L and Java languages;
2. Communication among distributed agents and;
3. Modular definition of communication protocols and other types of interaction, through their encapsulation within artifacts.

The communication infrastructure is composed by Communication Artifacts, which are divided into two groups: protocol artifacts, which encapsulate the logics of some communication protocol (a kind of communication that is, in general, more complex) and speech act artifacts, which execute simple speech acts.

Communication is achieved by executing operations available on these artifacts. Since they operate as communications mediators, their function is to route any message to its respective receivers and to supervise the order in which they are sent, if protocols are used.

The two basic classes of the communication infrastructure are: SpeechActArtifact and ProtocolArtifact. The former class defines the basic behavior of a speech act artifact, used for simpler communications. It manages the ongoing conversations that use artifacts of this type, storing the participants of the conversation as well as the message queues of each one. The latter class defines the basic behavior of a protocol artifact, used for more complex communications. This class concentrates the details involved in a protocol-based conversation, such as the respective automata (used to control the protocol) and conversation states for each agent involved.

The speech act artifacts are used simply to send messages, while the protocol artifacts are used to define an authorized sequence of messages to send (that is, defining the protocol’s execution flow). We remark that, in this way, the logics of protocols gets encapsulated in the corresponding protocol artifacts. On the other hand, in the speech act artifacts, just the communication performatives are defined. If other types of communication are used (for example, when message exchanges are not used), the communication infrastructure still allows such scenario (for example, when considering implicit communication using the environment). In this case, it is sufficient to create the artifact that mediates this communication, and another one related to the protocol that controls the execution flow. Currently, those artifacts do not exist in the infrastructure, but it is simple to extend it and include this type of communication.

The operations in the Communication Artifacts have the same name of the performatives they correspond to (e.g., an operation called inform sends an inform message), i.e., there is a direct mapping between the name of the operations and the messages they send. Moreover, agents use two basic operations:

1. subscribe: inserts the agent’s name and its CArtAgO Id into the artifact, being necessary to receive any exchanged message;
2. getMessage: recovers a message from the agent’s message queue;

4.1 Creating Communication Artifacts

To create a speech act artifact, a class that extends the SpeechActArtifact class must be created, implementing only the operation that executes that speech act.
For instance, an artifact that sends a message of the type Request needs only to implement an operation that sends this message. This operation should allow linking by other artifact if protocols are used.

The code fragment presented in Fig. 1 shows an artifact responsible by the Request performative. Its class extends the SpeechActArtifact class and its main operation is called request, having a conversation identifier, a receiver agent name and a message as parameters. The request operation defined in this artifact identifies the sender of the message and verifies if the receiver is registered in the artifact. If true, the message is inserted into its message queue (by the delivery method) and a signal is sent to this agent, informing that a new message has arrived. Otherwise, the operation fails and the sender agent is informed that its addressee is not registered.

```java
package speechActArtifacts;
import cartago.*;
public class FIPARequestArtifact extends SpeechActArtifact {

@OPERATION @LINK
//operation by which the communication act is performed
void request(String conversationId, String receiver,
  String message){
  String sender = getOpUserName();
  if(!nameToAgentId.containsKey(receiver)){
    System.out.println("Unknown agent on request artifact");
    failed("Unknown agent");
  } else{
    delivery(FIPAMessage.REQUEST, conversationId,
      receiver, message);
    signal(this.nameToAgentId.get(receiver),
      "new" + FIPAMessage.REQUEST + "Message", 
      sender, conversationId);
  }
}
}
```

Fig. 1. Artifact for the Request performative

The process to create a new protocol artifact is very similar to that of creating a speech act artifact. However, the complexity of such artifact is higher. Basically, a class extending the ProtocolArtifact class must be created. Then, all the automata related to the protocol's execution must be created and maintained, for verification of the permission to send each message of it. All the operations that send messages must have as a parameter the conversation identifier, regarding the current conversation. Also, it is necessary to keep a registry of all the ongoing conversations. It must be checked if the receiver agent is al-
ready included in the specified conversation, since new agents can be included in ongoing ones.

Figure 2 shows a class diagram, regarding the SpeechActArtifact and ProtocolArtifact classes, illustrating the creation of two new artifact instances from those types: the FIPAContractNet and FIPARequest artifacts, based on the homonymous protocol/performative defined in the FIPA-ACL language.

![Diagram illustrating how to create new communication artifacts](image)

**Fig. 2.** Diagram illustrating how to create new communication artifacts

### 4.2 How to use the Communication Infrastructure

First the artifacts must be created and this is done by an agent. By default, every artifact created in CArtAgO stays in the workspace called *default*. If it is desired a specific workspace to put the artifacts, it must be created through the `createWorkspace` operation. Afterwards, the agent must join it, using the `joinWorkspace` operation. To create an artifact, an agent calls the `makeArtifact` operation, specifying the complete name of the artifact's class. All those operations are provided by default in CArtAgO.

If only speech act artifacts are used, the agent simply creates then and can start using. However, if protocol artifacts are needed, after their creation the agent must call the `createArtifacts` operation, which will build all the speech act
artifacts used in the protocol to exchange its messages and link them to this artifact.

If the communication is remote (among JaCaMo instances and/or Java-based systems), the connecting nodes must know the IP address of the desired node and the name(s) of the workspace(s) were the artifacts are. If the connecting node is another JaCaMo instance, its agents only need to execute the external action `joinRemoteWorkspace` (provided by the CArtAgO framework), specifying the name and localization of the workspace where the artifacts are. If it is a Java node, it is necessary to initialize the CArtAgO node on the machine, then the agent that will use the artifacts must be started and the localization of the other node must be informed. Finally, the agent is able to use the communications infrastructure.

After the artifacts are created and all the initialization procedures are done, it is mandatory that the first action done by any agent is the execution of the `subscribe` operation. Then, for each artifact in use, it is necessary to execute the `focus` operation. Only then the agent can perceive whatever modifications in the observable properties of the artifacts and have them mapped to its belief base. Moreover, an essential functionality to use the infrastructure will become available: the `signal perception` (so that the agent can know when a message arrives, for instance). Finally, to exchange messages and perform communication, operations on the artifact are used. Each operation that sends a message triggers a signal, perceived by the receiver agent, informing that a new message has arrived as well as who sent it and the conversation identifier.

Since the communication infrastructure’s interface is described by signals and operations, it is possible to create a set of common plans (for AgentSpeak-L agents) and methods (for Java agents) for a preliminary treatment of each received message (e.g., identifying the message’s type and triggering an appropriate plan).

Figure 3 presents a code fragment showing how an `Inform` message can be treated by an AgentSpeak-L agent. The code shows that the agent perceives a signal, telling the message’s type, the sender and the conversation identifier. Based on this data, the agent decides what to do. However, the message is not recovered yet, the agent has to call the `getMessage` operation. Only then the message is removed from its message queue and delivered to the agent. For Java
agents, the process is very similar. Obviously, since it’s another language there are some minor changes, although the sequence of steps remains the same.

4.3 Examples on the use of the Communication Infrastructure

The simple examples presented in the following subsections are based on the modelling of a real world social system, the San Jerónimo Vegetable Garden (SJVG), a urban vegetable garden located in Seville (Spain), which is maintained by its own users under the supervision and coordination of the NGO “Ecologistas en Acción”. The harvest is entirely ecological, so no chemicals can be used. The production is only for self consumption, so it can not be sold.

The social organization of SJVG, considered as a urban ecosystem [12–14], is characterized for allowing a lot of interactions between the participants and have been used for experiments helped by MAS-based simulation [15, 16]. Among the roles identified in SJVG, in this section we consider the Ortolan (vegetable gardener), Auxiliar Ortolan and Technician.

(1) Technical Help by a Technician Agent In this example, two speech act artifacts are used, the Request and Inform ones.

There are two roles involved: the Technician and Ortolan. The agent who performs the Ortolan role creates the artifacts and starts the communication sending a Request message to the Technician agent, asking if he can use chemical fertilizer. The response to such event is an Inform message, containing the answer for the other agent’s request. In this case, the answer is "no". Figure 4 presents an activity diagram with the main steps followed in this example.

(2) Auxiliar Ortolan Binding Request In this scenario an agent performing the Ortolan role, requests to another agent performing the Secretariat role, an auxiliar ortolan binding. This process eases the harvest in the garden by binding other ortolans to help the currently participating ones. However, it is needed an authorization, so the agent has to ask it to the Secretariat role. This agent then checks whether or not this can be done and answers accordingly. Again the agent performing the Ortolan role creates the artifacts, while the Secretariat agent only answers the request. In this case, his answer is "yes" and an auxilar ortolan is bound to the requesting agent. Figure 5 presents an activity diagram with the main steps followed in this scenario.

5 Conclusion and Future Works

In this paper, a communication infrastructure for the JaCaMo platform was presented, based on the high-level artifact abstraction provided by the CArtAgO framework. Using the infrastructure, the JaCaMo platform gets another way of executing communication among its agents, enabling also communication for
agents implemented in different languages and distributed as well, using native JaCaMo resources, such as the ones provided by CArtAgO. Therefore, the communication infrastructure makes possible to abstract some specificities not related to communication i.e. the involved agents’ location and implementation languages.

The modular way of specifying interactions provided by the proposed infrastructure is its main contribution, since it is possible to define communication protocols and speech acts that are independent of the adopted organizational model and the implemented agents’ population. Also, since the artifacts’ approach is not exclusive to JaCaMo, the idea of using artifacts for communication can be applied on other scenarios as well. Encapsulating communication-related issues into artifacts makes them a common interface among all agents in the system, providing facilities to send/receive messages.

Finally, the proposed infrastructure only enables the execution of explicit communication (by exchange of messages) in JaCaMo. There are other types of communication to support, however this is a future work related to architectural improvements (to enable implicit communication, for instance). A performance analysis and a more accurate assessment of the infrastructure’s limitations will also be done, to add more communication-related facilities, such as whitepages.
A study of how this infrastructure can get connected to other JaCaMo dimensions (e.g. to relate roles to communication protocols) is another future work. Currently, the organizational model used in JaCaMo does not allow to describe interactions among roles. However, there are proposed extensions for it, as presented in [18], besides some other adaptations shown in [19]. There are other approaches, defining interactions in a new, disconnected from the organization and independent dimension, as in [17]. Figure 6 illustrates this last approach.

The definition of interactions in the organization shows more cohesion, so the connection of the infrastructure with an organizational model (such as MOISE+, already existing in the JaCaMo platform) will be studied. In Fig. 7 it is possible to observe this approach. It is intended to define interactions inside the system’s organizational specification, linking it to roles and goals. Therefore, the achievement of goals can be related to the utilization of protocols encapsulated in artifacts, for instance. However, this idea is in an early stage of development.

References


Embedding agents in business applications using enterprise integration patterns

Stephen Cranefield and Surangika Ranathunga
Department of Information Science, University of Otago, Dunedin, New Zealand
{scranefield,surangika}@infoscience.otago.ac.nz

Abstract. This paper addresses the issue of integrating agents with a variety of external resources and services, as found in enterprise computing environments. We propose an approach for interfacing agents and existing message routing and mediation engines based on the endpoint concept from the enterprise integration patterns of Hohpe and Woolf. A design for agent endpoints is presented, and an architecture for connecting the Jason agent platform to the Apache Camel enterprise integration framework using this type of endpoint is described. The approach is illustrated by means of a business process use case, and a number of Camel routes are presented. These demonstrate the benefits of interfacing agents to external services via a specialised message routing tool that supports enterprise integration patterns.

1 Introduction

Much of the research in multi-agent systems (MAS) is based on a conceptual model in which the only entities are agents and an abstracted external environment. This is in contrast to modern enterprise computing environments, which comprise a diverse range of middleware and server technologies.

The current solutions for integrating agents with external computing infrastructure are: (a) to access these resources and services directly from agent code (if using a conventional programming language), (b) to implement user-defined agent actions or an environment model to encapsulate these interactions, (c) to provide custom support in an agent platform for specific types of external service, or (d) to provide a generic interface for calling external resources and services, either using a platform-specific API [9] or by encapsulating them as agents [4], artifacts [8] or active components [7]. However, none of these approaches are a good solution when agents need to be integrated with a range of technologies. They either require agent developers to learn a variety of APIs, or they assume that agent platform developers or their users will provide wrapper templates for a significant number of commonly used technologies.

This paper proposes an alternative approach: the use of a direct bridge between agents and the mainstream industry technology for enterprise application integration: message routing and mediation engines, and in particular, those that support the enterprise integration patterns (EIP) of Hohpe and Woolf [5]. Our integration approach is illustrated in Figure 1. In this figure, each “pipes” graphic represents a messaging-based service coordination tool, such as an enterprise service bus [3]. The larger one represents an
organisation’s existing message-based infrastructure for managing business processes by coordinating information passing between applications and services. We propose that agents can be embedded into this infrastructure by integrating them with their own local message-routing and mediation engines, such as the lightweight Java-based Apache Camel enterprise integration framework [6]. This integration is based on the EIP notion of an endpoint, and we present the design of endpoints that can translate agent requests (encoded as agent communication language messages or action executions) to EIP messages, and from EIP messages to agent messages and percepts.

We describe an implemented architecture for connecting the Jason agent platform [2] to Camel using these “agent endpoints”. The approach is illustrated by means of a business process use case requiring the integration of Jason agents with a database management system, a mail server, a message broker and the Apache ZooKeeper coordination server. A number of Camel routes handling aspects of this use case are presented to demonstrate the benefits of interfacing agents to external services via a specialised message routing tool that supports enterprise integration patterns.

2 Enterprise Integration Patterns

Enterprise computing environments typically comprise hundreds and possibly thousands of applications [5]. These may use a variety of communication protocols and interface technologies due to departmental autonomy (e.g. to acquire “best of breed” applications for specific business problems), incremental and opportunistic growth, mergers, etc. To preserve loose coupling between the diverse applications involved in the automation of business processes, and thus facilitate maintenance and extensibility, the use of middleware products based on asynchronous message-passing has emerged as the mainstream approach for enterprise application integration. In this approach, applications interact

---

1 Pipes photo by Hervé Cozanet, source: http://commons.wikimedia.org/wiki/File:Piping_system_on_a_chemical_tanker.jpg (CC BY-SA 3.0)
by sending and receiving structured messages to and from named queues or publish-subscribe ‘topics’ managed by (possibly federated) message brokers. Message routing and transformation rules can be executed by the message broker or by specialised message routing and mediation engines, thus providing a single location for the specification of business processes. The concept of the enterprise service bus extends this idea further by integrating message brokers with middleware for deploying and interacting with various type of service, such as web services [3].

Hohpe and Woolf [5] have identified 65 “enterprise integration patterns” (EIPs) for solving basic problems that commonly arise in messaging-based enterprise application integration, such as the scatter-gather pattern: “How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?” A number of middleware tools have direct support for these patterns, including Apache Camel.

3 Apache Camel

Camel is an open source Java framework for executing message routing and mediation rules that are defined using domain-specific languages (DSLs) based on Java and Scala, or by using XML configuration files. In the work reported in this paper we have used the Java DSL.

Camel is based on the EIP concepts of routes and endpoints. A Camel application comprises a set of route definitions. Each route receives messages from a consumer endpoint, and performs a sequence of processing steps on each message, such as filtering and transforming messages, before sending the processed messages to one or more producer endpoints. Endpoints can be “direct” links to other routes in the application (i.e. messages leaving one route may flow directly into another route) or they may represent connections to external resources and services. For example, a mail endpoint may be used as a consumer to receive messages representing unread mail in a specified account on a mail server, or as a producer that sends mail to a server. Camel has more than 130 different components defined to provide a variety of endpoint types. These enable sending and/or receiving messages to and from external resources such as files, databases, message brokers, generic web services, specific Amazon and Google services, RSS and Atom feeds, and Twitter. To enable this diversity of endpoint types, Camel’s concept of a message is very general: a message has headers (a map from names to Java objects), a body (which can be any Java object) and optional attachments.

The code below defines two simple Camel routes. These use the agent component described in this paper to enable “local” agents (those running within the same process as the Camel routes) to communicate with remote agents via a message broker.

```java
from("agent:message")
  .setHeader("CamelJmsDestinationName",
    simple("$headers.receiver.split(\"\_\")[0]"))
  .to("jms:dummy")
from("jms:"+containerId).to("agent:message");
```
These routes are defined using Camel’s Java DSL. This is a Java API for constructing routes via a sequence of method calls. The from method creates a consumer endpoint and the to method creates a producer endpoint. Endpoints are specified using uniform resource identifiers (URIs), with the first part of the URI (the scheme) identifying the type of the endpoint. Other parts of the URI provide additional details, and the various endpoint types provided by Camel make use of URI parameters to provide configuration details for the instantiation of the endpoint. The routes shown above use two types of endpoint: the agent endpoint described in this paper, and Camel’s JMS endpoint for sending and receiving messages from a message broker using the Java Message Service.

The first route definition above creates an endpoint that receives all messages sent by local Jason agents. For each Jason message received, this endpoint copies the message content into the body of a new Camel message, and records the other message details using sender, receiver and illoc_force headers (these correspond directly to Jason message properties).

The routes are run within a Camel context object. Our architecture allows multiple distributed Camel contexts, each with their own set of local agents running within an agent container, so all agents are created with names of the form containerId__localName. The second and third lines of the first route above use Camel’s “Simple” expression language to extract the first part of the name, which identifies the agent container that the message recipient is attached to, and stores this as the value of a specific header predefined by the JMS component. When the message is processed by the JMS producer endpoint, this header is used to override the queue or topic name that appears as a mandatory component of a JMS endpoint URI (hence the “dummy” message queue name at the end of the first route above). This illustrates two aspects of the use of message headers in Camel: they are commonly used within routes to store information needed later in the route, and they can affect the handling of messages by endpoints.

The second route definition above creates a JMS endpoint that receives messages from a message broker (the address of the broker is provided to Camel’s JMS component on initialisation). The endpoint listens to a specific queue, which is named after the unique identifier for the local agent container (note that there may be agent containers associated with other Camel contexts running elsewhere on the network or in other processes). The JMS consumer endpoint copies the body and the message headers from the received JMS messages to create Camel message objects. The route specifies that these messages flow from the JMS consumer endpoint directly to a Jason producer endpoint. This endpoint generates Jason messages corresponding to the Camel messages and delivers them to the appropriate agents. The Jason producer endpoint does the reverse of the Camel to Jason message mapping described above.

Camel supports a number of message exchange patterns (MEPs), with the most commonly used being InOnly and InOut. The pattern to use for handling a message arriving at a consumer endpoint is set by that endpoint, possibly based on information in the message (such as a JMSReplyTo header on incoming JMS messages). The MEP can also be manually set by a route using methods of the Java DSL. If a message reaches the end of a route with the InOut MEP, it is returned to the consumer endpoint. If that endpoint supports it, that message will be treated as the reply to the initial request. Thus
Camel can be used to implement both synchronous and asynchronous processing of messages.

Note that Camel routes can be significantly more complicated than those shown above, as later examples in this paper will demonstrate. In particular, the Java DSL includes methods for conditional branching, exception-handling and for starting, stopping, suspending and resuming routes. In addition, an important feature of Camel is the provision of methods that can be used singly or in combination to implement enterprise integration patterns such as splitting and aggregating messages, or to “enrich” messages with content obtained by making synchronous calls to other endpoints.

4 A Jason/Camel bridge

In this section we briefly describe the architecture of our Jason/Camel bridge and discuss how we map between the conceptual models of Jason and Camel. In particular, we describe the design and interpretation of agent endpoints.

4.1 Application architecture

Our Jason/Camel bridge\(^2\) consists of an “agent component” for Camel and an application template that integrates the Jason BDI interpreter with a Camel context. The agent component for Camel is a factory for creating agent consumer and producer endpoints. Its implementation consists of the component class and classes that are instantiated to create producer and consumer endpoints for Jason.

The architecture of the bridge is shown in Figure 2 using UML\(^3\). A Camel application initialises any required components, creates a CamelContext object, passes it a RouteBuilder object with a method that defines the routes, and then starts the context. Our

\(^2\) [http://github.com/scranefield/camel-agent](http://github.com/scranefield/camel-agent)

\(^3\) Classes and interfaces developed or adapted by us are shaded in the figure.
integration architecture extends this by adding to the application an agent container. On initialisation, this container locates all Jason agent source (.asl) files in a given directory⁴ and, for each agent, instantiates our extension of the SimpleJasonAgent class⁵. This class allows the Jason BDI interpreter to be used without any of the existing Jason “infrastructures” for agent communication. It is responsible for providing the BDI interpreter with methods to call to get percepts, to perform actions, and to send and check for messages. We chose this as the most lightweight approach for embedding Jason agents into business processes via Camel

Our SimpleJasonAgent class maintains concurrently accessible queues for percepts of two types (transient and persistent) and for incoming messages. Messages on these queues are read (and consumed in the case of transient percepts) when the BDI interpreter calls the class’s methods for getting percepts and messages. Note that each agent and endpoint runs in a separate thread. The agent container writes messages and percepts to the queues for the relevant agents after receiving them from agent:message and agent:percept endpoints that appear in Camel routes. An endpoint for producing percepts chooses whether percepts are transient or persistent based on the endpoint URI parameters and/or the headers of the Camel message being processed. Transient percepts are cleared after an agent has perceived them, whereas persistent ones will repeatedly perceived (but may be overwritten by other percepts with the same functor—see the discussion of the updateMode URI parameter and message header in Section 4.2).

On construction, each agent is passed a list of agent consumer endpoints, and these are used to deliver messages and actions—the endpoints are responsible for selecting which of these match their configuration parameters. Camel messages generated by the consumer endpoints are processed using the InOnly message exchange pattern, unless specified otherwise by a route or an endpoint URI.

Inter-agent messaging via a message broker, as implemented by the routes shown above in Section 3, requires the existence of a separate message queue for each agent container. To enable this functionality, our application class has a optional configuration parameter specifying that an Apache ZooKeeper⁶ server should be used to dynamically obtain a unique identifier for the container.

A ZooKeeper server maintains a set of named nodes, arranged in a tree structure, to which system configuration information can be read and written by clients. The nodes are kept in memory to enable high performance, but transaction logs and persistent snapshots are also used to provide reliability. The data can be replicated across a cluster of ZooKeeper servers. Nodes can be persistent or ephemeral—a node of the latter type is automatically deleted if the client session that created it is no longer maintaining a “heartbeat”. Nodes can also be sequential. These have a unique number appended to the specified node name, based on a counter associated with the parent node. A client can place a watch for changes to the data recorded in a node, the existence of a node, or the set of children of a node. Together, these features can be used to implement a range

⁴ This simple approach will be replaced in the future by the use of OSGi “bundles” to package and deploy Camel contexts together with their associated agents.
⁵ http://jason.sourceforge.net/faq/faq.html#SECTION00057000000000000000
⁶ http://zookeeper.apache.org/
**Consumer endpoints**

<table>
<thead>
<tr>
<th>Endpoint type</th>
<th>Optional parameters</th>
<th>Camel headers set</th>
<th>Camel body contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent:message</td>
<td>illoc_force, sender, receiver, annotations, match, replace</td>
<td>illoc_force, sender, receiver, annotations, msg_id</td>
<td>The message content (as a string)</td>
</tr>
<tr>
<td>agent:action</td>
<td>actor, annotations, match, replace, resultHeaderMap</td>
<td>actor, annotations, actionName, params</td>
<td>The action term (as a string)</td>
</tr>
</tbody>
</table>

**Producer endpoints**

<table>
<thead>
<tr>
<th>Endpoint type</th>
<th>Optional parameters</th>
<th>Camel headers used</th>
<th>Camel body expected to be</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent:message</td>
<td>illoc_force, sender, receiver, annotations</td>
<td>illoc_force, sender, receiver, annotations</td>
<td>The message content (as a string)</td>
</tr>
<tr>
<td>agent:percept</td>
<td>receiver, annotations, persistent, updateMode</td>
<td>receiver, annotations, persistent, updateMode</td>
<td>The percept (as a string)</td>
</tr>
</tbody>
</table>

Table 1. Agent endpoint types

of distributed coordination mechanisms, such as distributed queues, barriers and locks, maintaining lists of active group members, and electing group leaders.

Our application class obtains the agent container identifier by requesting the creation of an ephemeral sequence node with the path `containers/container` and receives in response the name of the created node with a sequence number appended.

ZooKeeper servers can also be accessed from within Camel routes, via ZooKeeper endpoints. A use case for this functionality is illustrated in our MAS application scenario in Section 5.

Another option provided by our bridge is to directly deliver messages between agents that are in the same context, if preferred, rather than sending these to the Camel context for routing via JMS or any other means specified by the provided routes.

### 4.2 Agent endpoint design

We support two types of Jason consumer endpoints to handle local agent messages and actions delivered to them from our Jason/Camel bridge. Endpoints of these types generate Camel messages that correspond (respectively) to messages sent by the local agents and to actions executed by them. The details of the Jason messages and actions are encoded in the headers and body of the Camel message, as shown in Table 1. For example, the content of a Jason message is placed in the body of the Camel message, and the illoc_force (illocutionary force), sender, receiver, msg_id and
A route definition creates these types of endpoints by calling the `from` method with an argument that is a string of the form "agent:message?options" or "agent:action?options". The options are specified using the standard URI query parameter syntax `?opt1=v1&opt2=v2`. Camel messages are only generated by these endpoints if the selection criteria specified by the optional parameters are satisfied. The parameters recognised by these endpoint types are shown in Table 1 and explained below.

We also support two types of Jason producer endpoints, which generate messages and percepts, respectively, for the local agents. These messages and percepts are created from Camel messages that reach the endpoints via Camel routes, and their content is taken from the body and headers of those Camel messages and the endpoint URI parameters. As shown in Table 1, the URI parameters supported for the producer endpoints are mirrored by the headers that the endpoints check. This is because these message headers can be used to override the URI parameters when converting a Camel message to a Jason message or percept. This allows Camel routes to dynamically control the delivery and construction of Jason messages and percepts.

The URI endpoint parameters and Camel message headers are used as agent message and action selectors (for consumer endpoints) or to specify generated percepts or agent messages (for producer endpoints). Below, we provide some additional details for some of the parameter and header options.

**receiver:** We interpret the value “all” for this URI parameter and message header as meaning that only broadcast messages should be selected by a message consumer endpoint or that the message should be sent to all local agents from a message or percept producer endpoint. This is the default value for a producer. No agent can have this name because the agent container identifier is prepended to the names of all agents on creation. The `receiver` value can also be a comma-separated list of recipients when provided to a message or percept producer endpoint.

**annotations:** Jason supports the attachment of a list of annotation terms to a literal. An `annotations` URI parameter or header can be specified for controlling the selection of messages or actions by a consumer endpoint or to trigger the generation of annotations by a producer endpoint. The values are specified as a comma-separated list of literal strings (for the parameter) or as a Java list of strings (when using a header).

**match and replace:** These are used on consumer endpoints. A `match` parameter specifies a regular expression, and a Camel message is only generated if this matches the incoming message or action (in string format). The Java regular expression syntax is used, and pairs of parentheses may be used to specify ‘groups’ in the pattern. The values corresponding to these groups in the matched string are recorded and used when processing a `replace` parameter (if present). A `replace` parameter specifies a string to be used as the body of the generated Camel message. This can contain group variables (in the form `$n`), and these are replaced with the values that were recorded during matching.

**resultHeaderMap:** An action consumer endpoint supports both synchronous and asynchronous actions. An asynchronous action corresponds to a Jason external action
(which cannot contain variables), and the endpoint always returns the result true to the agent that performed the action. In order to handle actions that map to routes with an InOut message exchange pattern, we implemented a Java class that provides a new Jason internal action jasoncamel.syncInOut-Exchange. This is used to send terms that represent actions with unbound variables to Jason action consumer endpoints. Once the route (which is run with an InOut message exchange pattern) is completed, the endpoint unifies the variables in the action term with the resulting Camel message body. An endpoint processing this type of action must have a resultHeaderMap endpoint parameter. Its value should be a comma-separated list of header-name:argument-index pairs. When a Camel message completes the route, for each of these pairs the value of the header with name header-name header is unified with the action term’s argument at index argument-index.

**persistent and updateMode:** As described in Section 4.1, percepts delivered to agents by a percepts producer endpoint can be transient or persistent. The choice is controlled by the updateMode URI parameter or a message header with that name. Persistent percepts with the same functor and arity but different argument values can either accumulate in an agent’s persistent percepts list, or each new percept of that form can replace previous ones. The latter case is useful for percepts that represent the state of an external resource. A value of “-+” for an updateMode URI parameter or a Camel message header with that name can be used to specify the percept replacement behaviour. This can also apply to transient beliefs to prevent multiple percepts with the same functor and arity being queued up between consecutive perceptions by an agent.

### 5 A business process use case

In this section we illustrate the use of the Jason/Camel bridge by describing a hypothetical business process in which agents could play a valuable role. This use case addresses the problem of achieving more targeted information flow within an organisation and
reducing the overuse of the CC header in email messages. Our solution, shown in Figure 3, assumes the existence of a specific “to.share” email account. Users with information they think may be of interest to others can mail it to this account. Agents monitor this account and evaluate the relevance of each new message to other users, with each agent responsible for considering the interests and needs of a specific subset of users. The sets of users assigned to the agents form a partition of the complete user base. The agents base their decision on knowledge of the roles of users and the organisational structure (stored in a database), as well as specific plans that may optionally be provided by users to encode their goals for receiving information. We assume that these plans are created using a graphical web interface that provides end users with an abstraction layer on top of Jason’s plan syntax. When agents determine which users might be interested in an email message, they deliver the message to those users’ mail accounts via SMTP.

Our system design for implementing this business process involves coordinated use of Jason agents, a mail server, a database management system, a message broker and ZooKeeper, with the coordination performed by Camel routes. The key routes are as follows\(^7\).

1. On start-up, each agent performs an action get_email_accounts(Accounts) that is mapped to a database query by a Camel route. The route sets a message header to hold the list of accounts, and the agent consumer endpoint instantiates the argument Accounts with this value. After this action succeeds, the agent records this account list in a belief. This route is shown in Listing 1.

2. On start-up, each agent also performs a register action. A route maps this to the creation of an ephemeral sequential node in ZooKeeper (under the node /agents).

3. A route is watching the children of the ZooKeeper node /agents. Whenever there is a change (due to Camel contexts and their associated agent containers starting and stopping), the route sends an updated list of active agents to its local agents as a persistent percept in ++ update mode. This, and the route described in the previous paragraph, are shown in Listing 2.

4. Whenever a new email account is created or deleted by the administrator, the database is updated, and in addition a notification of the change is sent to a specific publish-subscribe topic on a message broker (a topic is needed rather than a queue

\(^7\) Note that the two example routes presented earlier in Section 3 are not used because agents in this application do not send messages to each other, but rather interact with external services via actions and percepts.
to allow *all* running Camel contexts to receive the message). A route monitors this topic and sends any received messages as transient percepts to all local agents.

5. Similar routes are provided for agents to obtain from the database information about users’ roles and their positions in the organisation structure, as well as a set of default email-forwarding plans. This is done by database queries from routes that are triggered by agent actions. Notifications of changes to this information are sent by an administrative tool (or a database trigger) to a message broker topic. A route monitoring that topic generates updates to the corresponding persistent agent percepts, which then cause agents to call the actions to load this information again.

6. Similar routes are also provided for agents to retrieve from a database, for a specified set of users, their information relevance assessment plans, and to receive notifications of changes to these plans via a message broker topic.

7. The agents have plans that react to changes in their beliefs about the currently active agents and the list of email accounts. When a change occurs, they each run an algorithm (common to all agents) to divide the list of email accounts amongst them, based on their own position in the list of agents. They maintain a belief recording the accounts they are responsible for.

8. A set of routes polls the “to.share” email account for new mail using a mail consumer endpoint, sends a message to all local agents asking them to evaluate which of their allocated email accounts the message is relevant for, aggregates the reply with the email message, and forwards it to the nominated users, via a mail producer endpoint. These routes are shown in Listing 3.

9. When the list of accounts that an agent is responsible for changes, or it is notified of changes to the plans for any of the accounts it handles, it must re-fetch plans for the relevant agents. The routes handling these notifications suspend the mail-polling route and start another route that uses a timer endpoint to resume the mail-polling route after a fixed amount of time. This gives agents time to fetch any required new plans from the database.

Listings 1, 2 and 3 show the routes we have implemented and tested for three aspects of the system’s functionality. We underline the beginnings of the agent endpoint URIs to highlight where the integration with agents occurs. Listing 1 shows how the execution of an agent action literal with a free variable can be implemented by a Camel route with the InOut message exchange pattern (note the use of the standard Camel URI parameter exchangePattern). The route sends an SQL query to a pre-configured database connection, the returned result is converted to an AgentSpeak list of strings using a Groovy expression, and then the result header is used to store the result. The consumer endpoint URI has a resultHeaderMap parameter specifying that the endpoint should unify the value of the result header with the first argument of the action literal.

Listing 2 illustrates how Camel provides a convenient way to use ZooKeeper to monitor the active members of a distributed group of agents, and to map this information to agent percepts. The first route (lines 3–11) implements the agent register action by creating an ephemeral sequential node (see Section 4.1) in a ZooKeeper server to represent the agent, and storing its name in that node. Camel’s support for the idempotent receiver enterprise integration pattern provides a simple way to filter out duplicate
Listing 2. Camel routes for tracking active agents via ZooKeeper

```java
// Implement registration by creating a new ZooKeeper sequence
from("agent:action?actionName=register")
// node with the agent name as its content
+ header("actor"),
MemoryIdempotentRepository.memoryIdempotentRepository(100)
.idempotentConsumer()
.to("zookeeper://" + zkserver + "/agents/agent" + "?create=true&createMode=EPHEMERAL_SEQUENTIAL");
// Process only one register action from each agent
.idempotentConsumer()
// Put actor name in message body
.setBody(header("actor"))
.to("zookeeper://" + zkserver + "/agents/agent" + "?create=true&createMode=EPHEMERAL_SEQUENTIAL");
// Watch agents node in ZooKeeper for changes to list of children
from("zookeeper://" + zkserver + "/agents/agent" + "?listChildren=true&repeat=true")
.setHeader("numChildren", simple(${body.size}))
.split(body())
// Split agent node list into separate messages
.process(new Processor() {
    public void process(Exchange exchange) throws Exception {
        // Map the ZooKeeper node name for an agent to the agent
        // name by getting the content of the ZooKeeper node
        ConsumerTemplate consumer = camel.createConsumerTemplate();
        String agentName =
            consumer.receiveBody("zookeeper://" + zkserver + "/agents/agent" + "?listChildren=true&repeat=true")
                .getBody()
                .asString();
        exchange.getIn().setBody(agentName);
    }
})
// Aggregate mapped names into a single message containing a
// list of names. All messages will have the same headers — any
// will do as the message correlation id
.aggregate(header("numChildren"),
    new ArrayListAggregationStrategy())
.completionSize(header("numChildren"))
.setBody(simple("agents(${bodyAs(String)})"))
.to("agent:percept?persistent=true&updateMode=-+");
```
Listing 3. Camel routes for forwarding email based on agent recommendations

```java
// Poll for email messages
from("imaps://mail.bigcorp.com?username=to.share" + "&password="+mailPassword+"&delete=true&copyTo=processed")
  .setHeader("id", simple("${id}\"\")
  .to("seda:forward-message", "direct:ask-agents");

// Request agents to evaluate message on behalf of their allocated users
from("direct:ask-agents")
  .setBody(simple("check_relevance(" + "${header.id}, " + "${header.from}" + "\"${header.subject}\"", "${bodyAs(String)}")"))
  .setHeader("receiver", constant("all"))
  .setHeader("sender", constant("router"))
  .to("agent:message?illoc_force=achieve");

// Receive responses from agents and aggregate them to get a single lists of relevant users
from("agent:message?illoc_force=tell" + "&receiver=router" + "&match=relevant\((.*)\),\((.*).\)\)" + "&replace=\$1:\$2")
  .setHeader("id", simple("${body.split(\":\")[0]}\")
  .setBody(simple("${body.split("\":\")[2]}\")
  .aggregate(header("id"),
    new SetUnionAggregationStrategy()
  .setHeader("to", simple("${bodyAs(String)}\")
  .to("seda:forward-message");

// Aggregate original mail message with message summarising interested users in "to" header, and send it
from("seda:forward-message")
  .aggregate(header("id"),
    new CombineBodyAndHeaderAggregationStrategy("to")
  ).completionSize(2)
  .setHeader("from", constant("to.share@bigcorp.com"))
  .to("smtp://to.share@mail.bigcorp.com?password="+mailPassword");
```
registration requests from agents. The second route (lines 14–36) is triggered by changes to the set of ZooKeeper sequence nodes representing agents. On each change, it receives a message listing the current sequence nodes. The splitter pattern is used (line 17) to obtain a separate message for each node, and each of these triggers a query to ZooKeeper to get the agent name stored at that node (lines 22–27). Finally (lines 32–34), the aggregator pattern is used to combine the names into a list stored in the body of a single message, and that is sent to the local agents as the argument of a percept (lines 35–36).

In the first route in Listing 3, the to.share mail account is polled for new mail (lines 2–3). A Camel message representing each new mail message is generated and the Camel message exchange identifier is written to a message header for latter use in correlating the agent responses with this Camel message (line 4). The message is then forwarded to two other routes (line 5). One is started asynchronously (via a “seda” endpoint, which queues incoming messages) and the other synchronously (via a “direct” endpoint). The second route (lines 9–16) sends an achieve request to the local agents, asking them to consider whether the mail is relevant to any of their allocated users. The third route (lines 20–28) handles messages sent by agents in response to this goal, which contain lists of potentially interested users. The aggregator pattern (lines 24–26) is used to produce, for each email message, a single message containing a combined list of users to forward it to. This is sent to the final route (lines 32–37), which also has (in a queue) the Camel message containing the email message that is waiting to be forwarded. This route uses the aggregator pattern again to combine the email message and the list of users to send the message to (stored in the to header). Finally, an SMTP endpoint is used to send the mail to these users.

The routes discussed in this paper have been tested using Jason stubs and the necessary external services, but the full Jason code for this scenario has not yet been developed and is not the focus of this paper. However, because the coordination logic is factored out and encoded in the Camel routes, the agent code required will be much simpler than would otherwise be needed without the use of our Jason/Camel bridge. Most of the agent behaviour is to react to percepts sent from Camel by performing actions (e.g. to fetch an updated list of email accounts), and to use Jason’s .add_plan and .remove_plan internal actions to update the plans used to evaluate the relevance of email messages to users. In response to the goal to evaluate a message, the agent must call the user plans, collect the users for whom these plans succeed, and send these in a message to Camel. The agents must also recompute the allocation of users to agents whenever the set of agents changes or new users are added, which they detect via ‘new belief’ events.

6 Related Work

One of the oldest approaches to integrating agents with other technologies is the use of wrappers or transducers that make the functionality of all the tools to be interconnected available through agent communication [4]. The overall system coordination can then be treated as a pure multi-agent system coordination problem. However, this approach has not gained traction in industry and we do not see it as a viable approach for integrating agents into enterprise computing environments.
A pragmatic but low-level approach for integrating agents with external systems is to call them directly from the agent program. If an agent platform is a framework for using a mainstream programming language for agent development (e.g. JADE\(^8\)), then it is possible for agents to use whatever protocols and client libraries are supported in that language to invoke external services directly from within agents or to monitor for external events. An interpreter for a specialised agent programming language may allow user-defined code in the underlying implementation language to implement functionality called by the agent program. For example, new “internal actions” for Jason can be developed in Java, and these can use any Java communication libraries for external interaction. An agent’s environment abstraction is another potential location for user customisation. For example, a Jason developer can implement an environment class that acts as a facade for external interaction.

The integration of agents with web services has been an important topic over the last decade, and some agent platforms provide specific support for this. For example, the online documentation for the JADE platform includes tutorials on calling web services from JADE and exposing agent services as web services, and the Jack WebBots [1] framework allows web applications to be built using agents.

More generally, it would be possible for the developers of an agent platform (or its community) to provide support for connecting agents to a range of external resource and service types. For example, the IMPACT agent platform [9] includes a module that provides a uniform interface for connecting agents to external services, with support for a small number of service types already implemented.

The A&A (Agents and Artifacts) meta-model extends the concept of an agent environment to include artifacts. These represent resources and tools with observable properties and specific operations that agents can invoke. These can be used to provide services internal to an MAS, or as an interface to external services, such as web services [8]. However, it is unlikely that the developer and user community for any agent-specific technology, whether a specific platform like IMPACT or a more general approach such as A&A, could rival the scale and diversity provided by a more mainstream integration technology such as Camel, which supports more than 130 endpoint types. Also, for the case of A&A, an agent developer would need to learn multiple APIs (for each artifact type) when integrating agents with different types of external service. This is not the case in our approach (see Section 7).

The active components paradigm is a combination of a component model with agent concepts [7]. Active components can communicate via method calls or asynchronous messages and may be hierarchically composed of subcomponents. They run within a management infrastructure that controls non-functional properties such as persistence and replication. They may have internal architectures of different types, and this heterogeneity, combined with a uniform external interface model, facilitates the interoperability of different types of system that are encapsulated as active components. As with artifacts, the success of this approach for large-scale integration rests on the availability of active components encapsulating a wide range of service types. A Camel context could be encapsulated within an active component (or artifact). However, that would still require the concept of agent endpoints, as proposed here, plus replication of Camel’s API.

\(^8\) http://jade.tilab.com/
7 Conclusion

In this paper we have proposed a novel approach for integrating agents with external resources and services by leveraging the capabilities of existing enterprise integration technology. By using a mainstream technology we can benefit from the competitive market for robust integration tools (or the larger user base for open source software), and can have access to a much larger range of pre-built components for connecting to different resource and service types. This is evidenced by Camel’s large number of available endpoint types.

We presented the design of an interface between agents and the Camel integration framework in terms of the EIP endpoint concept. This can serve as a pattern for interconnecting agents with any type of message-based middleware.

We described an implemented architecture for this approach and illustrated its practical use in a hypothetical (but, we think, plausible) business process use case. The Camel routes we presented demonstrate the benefits of using a specialist coordination tool such as Camel for handling the coordination of distributed agents and services, and leaving the agent code to provide the required core functionality. This division of responsibilities also enables a division of implementation effort: the coordination logic can be developed by business process architects using a programming paradigm that directly supports common enterprise integration patterns, and less development time is needed from (currently scarce) agent programmers. An agent programmer using our framework does not need to learn any APIs for client libraries or protocols—the agent code can be based entirely on the traditional agent concepts of messages, actions and plans. The developer of the message-routing logic does not need to know much about agents except the basic concepts encoded in the agent endpoint design (message, illocutionary force, action, percept, etc.) and the syntax of the agent messages to be sent from and received by the message routes.

References

An Infrastructure for the Design and Development of Open Interaction Systems

Daniel Okouya¹, Nicoletta Fornara¹, and Marco Colombetti¹,²

¹ Università della Svizzera Italiana, via G. Buffi 13, 6900 Lugano, Switzerland
{daniel.okouya,nicoletta.fornara,marco.colombetti}@usi.ch
² Politecnico di Milano, piazza Leonardo da Vinci 32, 20135 Milano, Italy
marco.colombetti@polimi.it

Abstract. We propose an infrastructure for the design and development of Open Interaction Systems (OISs), based on solutions from the areas of Service Oriented Architecture, Semantic Technologies and Normative Multiagent Systems, in particular the OCeAN metamodel of Artificial Institution. OISs are open to diverse types of participants (i.e., software agents), and enable them to interact with each other to achieve their objectives. To do so the participants are allowed to interact in compliance with previously agreed-upon regulations provided by the system and on the basis of the semantics of the communicative acts performed, both of which are enforced by the system. The infrastructure we propose involves four layers: (i) the Messaging Layer, which enables observable ACL message exchanges between heterogeneous participants while respecting ownership boundaries; (ii) the Core Service Layer, which enables the participants with performing observable non-communicative actions relevant to the ongoing application (iii) the Bridging Layer, in charge of interpreting the participants’ actions in a form suitable for regulation; and (iv) the Regulation Layer, which holds the regulations and enforces them with respect to the participants’ activities.

Keywords: Open Interaction Systems, Artificial Institution, Ontologies, Normative Systems, Agent Communication

1 Introduction

Open Interaction Systems (OISs) are distributed systems which diverse types of participants (i.e., software agents) can freely join with the goal of interacting with each other to achieve their objectives. To do so the participants are allowed to interact by exchanging messages with rigorously defined syntax and semantics, in compliance with previously agreed-upon norms provided by the system; both the norms and the communication language are enforced by the system.
In our past work we have proposed the OCeAN metamodel [12] for the specification of OISs. In this paper we describe an infrastructure, currently under development, for the actual implementation of such systems. In designing this infrastructure we aim at guaranteeing openness and interoperability, while keeping as close as possible to those technologies that are sufficiently mature and stable, and are already adopted by a large industrial community. Among such technologies we include standard Service Oriented technologies [4] and Semantic Web technologies [14].

The infrastructure we propose involves four layers: (i) the **Messaging Layer**, which enables heterogeneous participants to interact with each other through communicative actions while respecting ownership boundaries; (ii) the **Core Service Layer**, which allows the participants to exploit the support services offered by the OIS to perform non-communicative actions; (iii) the **Bridging Layer**, in charge of interpreting the participants’ actions in a form suitable for regulation; and (iv), the **Regulation Layer**, which holds the norms regulating the interactions and enforces them relative to the participants’ actions. More specifically:

- the Messaging Layer provides a messaging protocol based on standard technologies (HTTP, SOAP, WSDL) and uses Web Service technologies for the transfer of messages between participants, by prescribing the use of a specific message transfer service exposed via WSDL; messages realize communicative or institutional acts and comply with OCeAN-ACL [10], an Agent Communication Language strongly based on Semantic Web technologies, and on OWL 2 DL in particular;
- the Core Service Layer makes certain complementary services available to the participants (e.g., an OIS realizing an e-marketplace may offer services related to payment, product delivery, and so on) to perform observable non-communicative actions relevant to the ongoing application;
- the Bridging Layer interprets the participants’ communicative and non-communicative actions in a form suitable for regulation. Coherently with the OCeAN metamodel, such acts either result into commitments (like in the case of acts of informing, requesting, etc.) or are regarded as attempts to perform institutional actions relying on suitable count-as rules;
- finally, the Regulation Layer realizes a normative context (again according to the OCeAN metamodel), that is, a set of artificial institutions specifying the institutional actions that can be performed and the set of norms that have to be followed.

In this paper we provide a detailed specification of all layers and describe the implementation, currently under development, of an infrastructure oriented to the implementation of an open e-marketplace. The paper is organized as follows. In Section 2 we describe the functionalities pertaining to the Messaging Layer and how we implement them by exploiting standard Web Service technology. In Section 3 we briefly sketch how the core services offered by the OIS can be actually realized, considering an e-marketplace as an example. In Section 4 we describe the functionalities pertaining to the Regulation Layer and how we implement them by exploiting Semantic Web technologies, and OWL ontologies.
in particular. In Section 5 we explain how relevant events taking place at either the Massaging or the Core Service Layer are made available to the Regulation Layer. In Section 6 we review some related works. Finally in Section 7 we draw some conclusions and briefly describe our plans for future work.

2 The Messaging Layer

In an OIS, a large part of the participants’ interaction is carried out through the exchange of suitable messages. Therefore the bottom layer of our infrastructure provides the means to enable heterogeneous participants to interact with each other by exchanging messages in a fully interoperable fashion. In addition, it does so in such a way that it ensures the observability of these interactions, to the purpose of regulation.

To this end our infrastructure integrates principles from Service Oriented Architecture (SOA) and from Multiagent Systems (MAS). First, a message transfer approach is prescribed that is neutral to the internals of the participants, and leverages standard technologies to facilitate widespread adoption. This is in contrast with approaches based on some of the most well known ready-to-use messaging technologies like JMS\(^3\), RMI\(^4\), and CORBA [17], which bind either to a particular programing language [13] or to a programing language paradigm [17]. Such approaches do not fully decouple the end point implementation from the messages, thus limiting interoperability [18, 2]. Our architecture, following SOA’s principles of loose coupling, solely prescribes a message format together with its transfer protocol, both of them strictly decoupled from the end point implementation, while insisting as much as possible on the adoption of standard technologies [5, 4, 21].

Next, we add to the architectural prescriptions, the combination of the SOA concept of a message, as comprised of carrying and content information, with the MAS idea of a powerful and flexible Agent Communication Language (ACL). More precisely, we take the content part of a SOA message to represent the various components of a suitably designed ACL. Thus, together with the neutral messaging protocol delineated above, the participants are enabled to interact through the performance of communicative acts, in a totally interoperable fashion.

Finally, to enable the observability of the communicative acts performed by the participants to the purpose of regulation, this layer further mandates a Communication Channel (CC) to mediate message exchanges between participants. More precisely, to communicate with other registered participants, a registered participant shall send its messages to the address of the CC, with the name of the desired participants as the recipients. The CC receives the message which, if approved by the regulative process of the infrastructure, is then delivered to the intended participants. If the message is not approved, the CC rejects it and sends a suitable explanation to the sender.

\(^3\) [http://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html]

\(^4\) [http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html]
These architectural requirements are met in the infrastructure as follows. In the first place, the infrastructure provides for a messaging protocol based on standard neutral technologies: HTTP, SOAP\(^5\), and WSDL\(^6\). In other words, Web Service technology is adopted for message transfer between participants, by specifying a message transfer service, exposed via WSDL, in which HTTP is used for the transport of messages and SOAP for their structure. Our choice is motivated by the fact that this technology represents a standard approach for making available over the network functionalities that are triggered or delivered by exchanging messages.

In the second place, the infrastructure then specifies the body of the SOAP messages as messages of our ACL\(^7\). From the syntactic point of view, the ACL we propose is very close to KQML\(^8\) and FIPA ACL\(^9\), from which however it substantially departs as far as semantics is concerned (see Section 5). As with FIPA standards, our ACL come with a separate Content Language (CL). Our CL is defined as an OWL Ontology, the Content Language Ontology\([10]\). It plays a role similar to FIPA-RDF.

Thus, realizing the first two requirements, we define a WSDL file with only one service, which is the delivery of an ACL message, carried in the body of a SOAP message. The WSDL contract represents any form of message that can be exchanged between entities of our OIS, with the requirement that the message contains the address to reply to according to the same contract. Communication between participants is only allowed through the use of this service; consequently, all participants are required to be equipped with a suitable communication module, composed of: (i), a *listening-point*, that is, a web-service provider exposing a message delivery service defined according to our WSDL contract; and (ii), a *talking-point*, that is, a web-service client that requests the delivery of a message in conformance with that contract \([9]\).

A crucial advantage of this approach is the provision of a messaging protocol in the form of a WSDL contract, which is both human readable and machine processable. Such a contract can be easily handled with the support of runtime frameworks coming along with Web Service technology, such as Apache CXF \([1,15]\). We use CXF to automatically generate the core of the communication module of the participating component of our infrastructure; hence anyone can easily generate the necessary facilities to handle the transmission of messages abiding to the exposed messaging protocol and adapt it to their need, in order to participate in the OIS.

Finally, to deal with message transfer the infrastructure provides an implementation of the CC as a Java component, developed with CXF as exposed above.

\(^5\) [http://www.w3.org/TR/soap](http://www.w3.org/TR/soap)
\(^6\) [http://www.w3.org/TR/wsd1](http://www.w3.org/TR/wsd1)
\(^7\) [http://www.people.lu.unisi.ch/okouyad/AclOverSoapHttpMP.wsdl](http://www.people.lu.unisi.ch/okouyad/AclOverSoapHttpMP.wsdl)
\(^8\) [http://www.csee.umbc.edu/csee/research/kqml/](http://www.csee.umbc.edu/csee/research/kqml/)
\(^9\) [http://www.fipa.org/repository/aclspecs.html](http://www.fipa.org/repository/aclspecs.html)
As we have already remarked, in OISs, a large part of the participants’ interaction is carried out by exchanging suitable messages; under the circumstances of our Infrastructure, as a result of its messaging layer, it can be further stated that it is mainly by performing communicative acts, that is, by exchanging ACL messages. However, most types of applications will also require the executions of actions that are not strictly speaking communicative. We identify these actions as *non-communicative acts* and classify them into two categories. First, non-communicative acts that concern the interaction between the participants and certain components of the infrastructure, designed to provide support to the participants’ activities; as we shall see, these non-communicative acts are typically *application-independent*. Second, non-communicative acts occurring between the participants that concern certain *application-specific* interactions.

More specifically, on the one hand, some of the *application-independent* non-communicative actions are intended to support the enforcement of ownership-boundaries between participants, enabling them to connect with each other without introducing dependencies. To this purpose, the infrastructure provides for a *Registry* component, within which the participants can be listed or unlisted by performing actions such as *registering* or *deregistering* their identities. Although the registration and deregistration processes do presuppose the performance of certain communicative actions (more precisely, of the request to be registered or deregistered), the actions of registering or deregistering a participant are not themselves communicative. Rather, they are non-communicative actions made available to the participants by the infrastructure, through the provision of services that may be invoked using communicative actions (requests).

On the other hand, some of the application-specific activities, that is, some of the activities that are carried out between the participants, may also require more than the sole performance of communicative actions. That is, the nature of the interactions may demand the performance of *application-specific* non-communicative actions, which, as in the case of communicative acts (i.e., the other actions occurring between the participants), must also be made observable to the infrastructure. For example, in an e-marketplace system, when engaging in a purchasing activity, after settling a contract with communicative acts, the buyer may be required to carry out a payment, while the seller may be required to deliver a product. These are both non-communicative acts inherent to the purchasing activity, and as such must also be visible to the infrastructure.

Thus, the objective of this Layer is to equip the infrastructure so that: (i), it enables the participants with performing all the infrastructure-specific *non-communicative actions* belonging to the direct interactions between the participants and the infrastructure itself; and (ii), it can observe the performance of the application-specific *non-communicative actions* inherent to some of the activities occurring between the participants.

To this end, first, as suggested above, for those non-communicative actions that are *application-independent* (i.e., infrastructure-specific) at present our infrastructure provides a Registry component, implemented in Java, to serve as a
White Pages Service. It provides among others, for the registering and deregistering actions. As this component is endowed with ACL-processing capabilities, participants can request its services using ACL messages.

Next, for the non-communicative actions that are application-specific, in unison with the approach used for the communicative acts (i.e., that the observation of the actions occurring between the participants goes through the mediation of their performance), the infrastructure also proposes to mediate them. In this respect, however, the core service layer proceeds differently from the messaging layer. Indeed, the different communicative actions that can be performed by the participants are the same across applications; thus, the observation process necessary to handle them is also application-independent, and therefore can be achieved by a generic component: the Communication Channel. In contrast, non-communicative acts occurring between participants are typically application-dependent: their presence, what they achieve, and how they achieve it, always depend on the application being realized. Indeed, on one hand, unlike communicative acts that ought to be always available to the participants, the presence of those non-communicative actions is application-specific; for instance, the availability of a delivery action would be irrelevant to an application that does not deal with delivery, such as an e-market for computational services.

On the other hand, when present, the performance of those non-communicative acts can substantially vary depending on the requirements of the applications in which they are performed: as illustrated by the case of a payment, while one application may require a system like PayPal, another one may require a direct bank-to-bank transfer or a cheque payment, which would require to go through different steps and to supply different information. Another important difference is that, unlike communicative actions, non-communicative actions can also vary in nature, that is, they can be electronic, physical or involve both aspects.

Hence, to mediate them, the infrastructure must proceed carefully taking into account their fundamental application-oriented characteristics, as well as their nature that can involve any combination of physical and electronic aspects. To achieve this, the architecture prescribes that the Core Service Layer provides for the incorporation of observable application-specific components, offering to the participants specific services of mediation for those application-specific, non-communicative actions. These components must be such that they seemingly interoperate with the participants for the invocation of the actions that they mediate, whose performances must be observable.

To that purpose, on the one hand, this layer specifies the interfaces of the mediating components, so that the relevant parts of the infrastructure can take into account the performance of the non-communicative actions they are in charge of. On the other hand, it prescribes the characteristics that the components must possess so that their services can be seemingly consumed. In support of that latter point, the layer mandates the use of communicative acts to invoke their mediation services. That is, while the message-transfer mediation service of the messaging Layer is invoked using a SOAP message (as a typical web-service), theses services are invoked using an ACL message, that is, the content of the
soap message. It brings the advantage of providing a unique flexible protocol for
the invocation of these services, independently of their nature and level of com-
plexity. This implies that those mediating components must be able to process
certain ACL messages.

Meanwhile, it is important to mention that as part of the service they pro-
vide, our infrastructure does not require that mediating components directly
perform the non-communicative actions they supply: indeed they may do so, or
guarantee their performance by some external systems, or simply acknowledge
their external realization as informed by a set participants who have agreed to
use an external service for their interaction. In this regard, the layer classifies
theses services into two distinct categories: internal services and external ser-
vice. In the former case, the service is internally managed by the component
itself; this means that when directly asked by a participant, the component takes
charge of the execution of the activities involved in the service. In the latter case,
which represent a very decentralized approach providing more freedom to the
participants, the execution is guaranteed by the participants themselves, which
then inform the infrastructure of the results. Here mediation plays the role of a
neutral authority that acknowledges the realization of services taking place out
of its direct control, according to the specific rules governing the application.

4 The Regulation Layer

Once heterogeneous participants, possibly belonging to different owners, can
interact with each other as exposed above, it is necessary that they get pro-
vided with some form of harnessing framework defining norms that regulate
their interactions. This is particularly important as it allows the participants
to have reasonable expectations with respect to the interactions they engage
in order to achieve their objectives. Moreover given that we target systems as
e-marketplaces, taking in account the sensitive nature of their activities, the ar-
chitecture prescribes the realization of a neutral third-party component in charge
of analyzing the participants’ interactions (by using the information received by
the Bridging Layer as described in Section 5), with the aim of monitoring the
 evolution of the state of the interaction and specifying and enforcing the norms
of the regulating context.

In order to realize all these functionalities we introduce in the proposed ar-
chitecture the Regulation Layer. It is based on the OCeAN meta-model [11],
in which regulating contexts are defined as artificial institutions that provide a
high-level representation of a specific set of institutional actions together with
the norms that govern them, and of the institutional objects that need to be
observed to monitor the evolution of the state of the interactions. For every spe-
cific application, such institutions are operationalized by grounding them in the
current domain [12, 7].

The Regulation Layer must possess a formal representation of the state of
the interaction suitable to carry out automatic reasoning. In particular this rep-
resentation has to include specifications of: (i), the regulating context in force;
(ii), the types of events and actions the application is dealing with; (iii), the application-dependent and application-independent knowledge defining the relevant objects and their states during the interaction; and (iv), the instances of the institutional actions and events that happen in the system. Reasoning will then allow the system to monitor the evolution of the state of the interaction, detecting in particular norms fulfillment and violation.

Our infrastructure meets these requirements in the following way. We define our regulating context as an OCeAN artificial institution. The first regulating context we have operationalized so far is the Commitment Institution, which regulates agent interactions in terms of the commitments they make to each other by the performance of communicative acts [6, 7]. This is an application-independent foundational institution, used in the definition of more specific application-dependent institutions (like for example the institutions formalizing different types of auctions). This Commitment Institution specifies commitments as institutional objects, together with their life-cycle rules and the institutional actions that allow an agent to create, cancel, or otherwise manipulate them. This enables us to monitor the state of an interaction in term of the evolution of the commitments that the participants make to each other. Application-dependent regulating contexts (like for example those relevant to e-commerce) are also represented as OCeAN institutions.

In our infrastructure, institutions as well as domain knowledge (e.g., knowledge about the products that are exchanged in the e-market) are represented as ontologies specified in OWL 2 DL [14], the standard language for defining ontologies in the Semantic Web. Also the state of an interaction is represented in an OWL ontology, that we call the Interaction Ontology, which is continually updated while the interaction proceeds (see Section 5). More precisely the Interaction Ontology contains a representation of the institutional objects defined by the institutions in force, along with when required, the institutional actions that create and manipulate them, altogether with the basic actions and objects the institution may operate with. To serve this purpose, the Interaction Ontology imports:

- an OWL upper ontology specifying common application-independent concepts like the notion of agent, action, event, and object;
- the SWRL Temporal Ontology\(^\text{10}\) for representing instants and intervals of time;
- the OWL ontologies used for representing the relevant artificial institutions e.g. the Commitment Institution Ontology\(^\text{11}\);
- the Domain Ontology used for representing relevant domain knowledge.

Some of these ontologies are described in details in [10]. The ontology imports are realized according to an architecture [10] that we have crafted specifically to avoid conflicts and duplications of those application-independent concepts (like agent, action, temporal interval, etc.) on which several ontologies overlap.

\(^{10}\) http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTemporalOntology

\(^{11}\) http://www.people.lu.unisi.ch/okouyad/CommitmentOntology.owl
Using OWL 2 DL reasoning, our representation makes it possible to monitor the state of the interactions according to the rules of the context. Thus, equipped with it, in compliance with the prescriptions of the architecture which require a neutral third-party component to enact this functionality, our infrastructure provides for a regulation component which plays the role of interaction manager, in charge of monitoring regulations and requesting their enforcement when necessary. To this purpose, the regulation component relies on the Pellet OWL 2 reasoner\textsuperscript{12} that it uses in conjunction with the OWL-API\textsuperscript{13}: every time a relevant event happens (such as the elapsing of a pertinent instant of time, or the realization of an institutional or non-institutional action or events), a suitable assertion is added to the ABox of the \textit{Interaction Ontology} and the reasoning process is triggered.

Implementing such a task by OWL 2 DL reasoning is not straightforward. First, as participants interactions have to be monitored over time, it is necessary to carry out some kind of temporal reasoning. For instance, if a participant has a commitment towards another to realize a given action before a deadline, in order to deduce that the after the deadline the commitment is fulfilled or violated it is necessary to deduce that the deadline has elapsed. This cannot be specified by OWL axioms alone; therefore, SWRL\textsuperscript{14} rules containing temporal built-ins have been added to perform suitable temporal inferences. Such rules exploit the Time Ontology developed by the Protege group [16], which provides a time representation format that is suitable for calculation, is aligned with the current XSD standards, and defines a rich set of temporal built-ins that can be used to extend our OWL ontologies with SWRL rules. However, given that these built-ins are not SWRL standards, they are not natively supported by reasoning engines; as the Protege group has provided an implementation for reasoning with these built-ins only with the Jess rule engine, we have developed our implementation for extending the reasoning capabilities of Pellet reasoner by using the Pellet custom built-ins definition mechanism.

Representing the evolution of the state of interactions (including for example the new commitments that the participants bring about) by means of a continuous update of the \textit{Interaction Ontology} at run-time [8], is a delicate task because it may introduce inconsistencies. More specifically, in our formalization of the \textit{Commitment Institution ontology}\textsuperscript{11} presented in [6], in which we refer to it as the \textit{Obligation ontology}, we specify that an action-commitment (i.e., a commitment to perform an action, namely, an obligation), has an associated temporal interval, within which the action must be executed. Determining this interval can involve several steps depending on the properties inherited by the the commitment at its creation. In certain situations such as when the action-commitment is conditional, it only becomes activated if a specific triggering event or action takes place; when this activation occurs, the beginning and the end instant of time of the interval associated to the action-commitment have

\textsuperscript{12} http://clarkparsia.com/pellet/
\textsuperscript{13} http://owlapi.sourceforge.net/
\textsuperscript{14} http://www.w3.org/Submission/SWRL/
to be set. For example, if the exchange of a message commits a participant to
deliver a product within two days, on condition that the receiver of the product
performs a payment, then the action-commitment will be created as soon as the
message is exchanged, but will only be activated when the payment takes place.
At activation time the interval will be determined as follows: (i), its beginning is
set at the time instant of the activation; and (ii), its end is set at the beginning
plus two days. All this can be expressed by a suitable SWRL rule. However,
if several actions belonging to the activation class of the obligation take place,
the SWRL rule will be activated several times and the interval of the obliga-
tion will be represented incorrectly. This problem cannot be solved inside the
OWL ontology, even by the use of additional SWRL rules; therefore we regulate
the activation of the relevant SWRL rule with an external Java program that
using the OWL-API, checks that an interval that is already set is not further
changed. In short, some reasoning and calculations have to be made outside of
the reasoner, in order to properly manage the Interaction Ontology.

5 The Bridging Layer

To regulate interactions it is necessary to capture the participants’ actions and
other relevant events that take place in the system, and to represent them in a
form that suits the abstraction level at which regulation operates. This is the
purpose of the Bridging Layer. For this, it prescribes a bridging component which
equipped with the definition of the institutions in force that are shared with the
regulation component, operates as detailed in the following.

First, all events (inclusive of the participants’ actions) that are relevant for
regulation must be observed by the Bridging component. These events take place
either at the Messaging Layer or at the Core Service Layer. As far as the former is
concerned, the relevant events consist in exchanges of ACL messages, which are
made available for observation by the CC (Communication Channel) component
of the Messaging Layer. To the purpose of regulation, it is therefore crucial that
all message exchanges between participants take place through the CC provided
by the infrastructure. As we have already remarked, however, message exchanges
are not the only events that need regulation. Among these also certain non-
communicative events are included, like for example the actions of payment or
delivery of products. These events are made available by the Core Service Layer.

Subsequently, the observed events have to be represented in a form that is
suitable for regulation. In particular, given that the Regulation Layer relies on
artificial institutions, representing an observed concrete event in a form suitable
for regulation involves producing a representation that is compatible with the
specification of the artificial institution.

In the OCeAN metamodel, artificial institutions deal with two types of
events, that we respectively call basic and institutional events. An institutional
event Y is an event that is brought about by the performance of another, lower
level event X, thanks to suitable counts-as rules, provided that certain enabling
conditions C hold. For example, an artificial institution may specify that a cer-
tain type of message sent by a suitably empowered agent A will count as an institutional action of opening an auction. Contrastingly, basic events are events that can be directly produced by a participant, without the need of realizing it through the performance of another, lower level event. For example, performing the concrete action of sending a message to another participant is represented in the institution as a basic event of message exchange.

Therefore, transforming an observed concrete event in a form suitable for regulation requires producing a representation of either a basic or institutional event. In the Regulation Layer, both artificial institutions and the concrete domains over which they operate are specified as OWL ontologies. Thus the infrastructure transforms the observed concrete event into OWL individuals that belong to classes of events pertaining either to the institution ontologies or to the concrete domain ontologies. More accurately, as institutional events are grounded on basic events, this transformation process consists of: (i), creating an OWL individual representing the basic event; and (ii), optionally creating an OWL individual representing the institutional event, if this is required by a count-as relationship defined in the institution in force.

Coherently with the OCeAN metamodel, we provide a set of application-independent counts-as links between message exchanges (considered as basic events) and the creation of suitable commitments (considered as institutional events): these rules are part of the Commitment Institutions and specify the application-independent component of the semantics of OCeAN-ACL. More specifically, according to the OCeAN-ACL semantics, the exchange of commissive messages (like promising) and directive messages (like requesting) are interpreted in the Commitment Institution as institutional actions that create action commitments [20], that is, commitments to perform the action described in the content part of the message. Commitments of this type can be considered as equivalent to obligations; for example, if agent A promises to agent B to pay a given sum of money M for a given product P, the communicative act will be interpreted as a create-obligation institutional action, that is, an attempt to create an obligation of agent A to pay M euros to B for product P. When the Bridging Layer delivers this institutional action to the Regulation Layer, the Interaction Ontology will be updated with a new institutional object of type Obligation, with A as the debtor, B as the creditor, and the payment of M euros for P as the content. Thereafter, the obligation will be monitored for its fulfillment, violation or cancellation as part of the process of interaction monitoring carried out by the Regulation Layer. Requests are treated in a similar way, except that they involve one more step; more precisely, a request is interpreted as the attempt to create an action precommitment (or preobligation), which in turn leads to an attempt to create an obligation for the receiver, if the receiver accepts the request (i.e., the preobligation).

Assertive communicative acts (like informing) are conceptually different from commissives and directives, because they introduce propositional commitments [20], which cannot be interpreted as ordinary obligations. For example, if agent A informs agent B that the product delivered is damaged, this commits A to the
truth of what is said (i.e., that the product is indeed damaged), but does not obligate A to perform any predefined action. We have not yet worked out a representation of propositional commitments for our infrastructure: this issue is therefore deferred to future works.

Finally, there is another type of communicative acts, which following the terminology of Searle’s Speech Act Theory [19] we call declarations; examples are declaring that an auction is open, or that a specific agent is the winner of an auction’s run. Declarations are carried out by exchanging suitable ACL messages, with declaration as the performative, and a content that represents the institutional action being performed. Coherently with the OCeAN metamodel, such messages are interpreted within an artificial institution through a counts-as rule, which generate the declared institutional action provided that certain conditions hold. Typically, a condition for the successful performance of a declaration is that the actor of the action has the institutional power to perform the declared institutional action (e.g., only an auctioneer can possibly open an auction). Such institutional powers are associated at design time to the different roles that can be played by a participant in an institution, and are checked at runtime by the Regulation Layer.

In practice, to achieve this transformation from basic events to institutional events, the OWL specifications of application-independent concepts (such as agent, action, event, object, time instant, time interval, etc.) are shared between the Content Language Ontology (see Section 2), the ontologies of the relevant institutions, and the domain ontologies over which the ongoing application operates and on which the institutions are grounded. The sharing is achieved thanks to the ontological architecture introduced in the Regulation Layer, which eliminates all the ontological mapping hurdles that would have otherwise been necessary to handle for the full transformation process to take place. Indeed it allows to seemingly go from one representation to another; for instance, going from the communicative action promise \( (A, B, \text{pay}(\text{book1}, 5 \text{ euro})) \) (which involves the Content Language Ontology and a concrete domain ontology) to the institutional action create-Obligation \( (A, B, \text{pay}(A, B, \text{book}, 5 \text{ euros}), \text{instant1}) \) (which involves the Commitment Institution Ontology and the same domain ontology) is achieved smoothly thanks to the underlying shared concepts of agent, action, object. If these concepts were not shared appropriately, mappings would have been necessary between the specifications of these concepts in different ontologies. The same principle applies, for example, when a non-communication action of payment happens that is represented by the OWL individual Pay\( (A, B, \text{book}, 5 \text{ euros, inst1}) \), which has to be transformed into the institutional action Acquire-Ownership\( (A, B, \text{book}, 5 \text{ euros, instant1}) \) of an hypothetical Ownership Institution (where the target representation is understood as \( A \) getting the ownership of \( \text{book} \) from \( B \), for the price of \( 5 \text{ euros at instant1} \)).
6 Related Work

Among the recent multiagent infrastructures focused on OISs, which in particular share the aim of providing the regulation of the participants’ interactions in the form of a neutral third-party functionality, as part of the overall support that they deliver, the Magentix2 Open multi-agent systems platform\footnote{http://www.gti-ia.upv.es/sma/tools/magentix2/\textsuperscript{15}}\cite{3} represents the state of the art on the matter. In particular it is the most advanced operational infrastructure, which includes many of the recent advances in the OIS area. We therefore provide a comparison with our infrastructure as a way to relate our work to the state of the art in the field.

At a very abstract level the two infrastructures share the same architectural approach. More precisely, although their respective concrete layered architecture are slightly differently structured, they present the same abstract architectural organization: a top part concerned with regulation specification and management, a bottom part concerned with the support of observable interactions between heterogenous participants, and a middle part concerned with the monitoring of the participants’ interactions according to the regulation in force and its enforcement when deemed appropriate. Consequently, differences only appears in the way the parts are concretely realized, with the most fundamental of them occurring in the middle part. This reflects a common vision of the role of the infrastructure, but divergences on how its different parts may concretely operate to achieve it.

More specifically, at the top level, Magentix2 adopts the metamodel of virtual organizations, which specifies roles with norms including platform generic roles such as OMS (Organization Management System) and DF (Directory Facilitator), for the specification of a regulation structure. Our infrastructure also defines a regulation structure at this level, but one that is based on the OCeAN metamodel of artificial institutions (see Section 4). While a thorough comparison of the two metamodels is outside the scope of this paper, it can be safely said that both infrastructure intend to provide similar regulating structures, which in particular are centered on non-regimented norms, to harness the participants’ activities.

At the bottom, both infrastructures provide an observable vehicle for the participants to interact with each other. To that end, they use similar approaches, but differ in the general understanding of interactions. Indeed the OCeAN metamodel classifies actions into communicative and non-communicative ones, which Magentix2 does not, in that it only considers communicative actions. Consequently, while we divide the bottom part of the infrastructure into two layers (Messaging and Core Service), with the upper one devoted to non-communicative actions and the lower one devoted to communicative actions, Magentix2 only provides one interaction level which corresponds to our lower layer.

As far as communicative interactions are concerned, the two infrastructures operate in a similar manner (as they both provide an end point neutral messaging protocol with a broker for interoperable communication between het-
erogenous participants), but diverge in the choice of the technology. Where we use WS (SOAP, HTTP, WSDL) with the SOAP Body structure defined as an OCeAN-ACL message for messages exchange, Magentix2 adopts AMQP\textsuperscript{16} with the message body structure defined as a FIPA-ACL message. We believe that the use of WS is more widespread and therefore easier to adopt than AMQP, which has yet to become a standard.

As previously mentioned, the sharpest differences between the infrastructures occurs in the middle part, whose functionality can be summarized as follows: (i), observing concrete events such as message exchanges or core-service events; (ii), representing observed events in a form suitable for regulation; (iii), checking them against the regulations for monitoring purposes; and (iv), enforcing the relevant regulations when deemed appropriate. It is with (ii) and (iii) that the two infrastructures differ substantially.

With our infrastructure, checking against regulations is done by means of reasoning over a representation of the state of the interaction, carried out within an OWL ontology that includes the institutions in force and the norms coming along. Our norms and their instantiations (in terms of obligations and prohibitions) are represented as OWL individuals, so that their activation, cancellation, fulfillment and violation conditions are represented as event types (i.e., as subclasses of class Event). Therefore we use the full power of DL reasoning to match the representations of concrete events with norms conditions. This process is much more powerful than the one adopted by Magentix2, which relies on the matching of a restricted subset of first-order logic formulas.

A further important difference between Magentix2 and our infrastructure is that the latter does not rely on an application-independent semantics of ACL messages. In our infrastructure, based on the OCeAN metamodel, the application-independent part of messages (i.e., all components of an ACL message with the exception of its content) is given a uniform semantics across applications. Moreover, such semantics allows for a representation of messages (produced by the Bridging Layer) that immediately relates message exchanges to the Regulation Layer. This means that only application-dependent non-communicative events will need to receive a special treatment in different applications of the infrastructure. Conversely, Magentix2 does not provide for any application independent connection between the participants’ actions and regulation, thus making the conversion to different application more expensive and error-prone.

7 Conclusions

In this paper we have presented an infrastructure for Open Interaction Systems, based on the OCeAN metamodel and currently under implementation. Our main concerns in the development of the infrastructure are, on the one hand, to guarantee openness and interoperability, and, on the other hand, to rely as much as possible on technologies that are sufficiently mature and stable, like Service Oriented and Semantic Web technologies, to facilitate adoption by the industry.

\textsuperscript{16} http://www.amqp.org/
The infrastructure has been divided into components to separate different concerns, which brings several advantages: on the one side, it enables us to distribute the infrastructure and to use techniques of dynamic adaptation (such as cloning and self-deletion) to manage overhead issues; on the other side it enables us to provide targeted upgrades and developments of the infrastructure. So far, for prototyping purposes the infrastructure is being implemented as a monolithic multi-threaded Java application. Nevertheless, the different components are present and well separated so that they can be easily extracted to provide a fully distributed infrastructure.

In the near future we intend to complete the implementation and test of the prototype. In particular we plan to complete the formalization in OWL of the semantics of the various type of communicative acts, to separate the various component of the prototype and to test it with the formalization and execution of an e-marketplace, inclusive of the OWL ontologies representing the relevant institutions and domain knowledge.

References

Engineering Pervasive Multiagent Systems in SAPERE

Ambra Molesini1, Andrea Omicini1, Mirko Viroli1, and Franco Zambonelli2

1 Dipartimento di Informatica–Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM–Università di Bologna, Italy
{ambra.molesini, andrea.omicini, mirko.viroli}@unibo.it

2 Dipartimento di Scienze e Metodi dell’Ingegneria (DISMI)
Università degli Studi di Modena e Reggio Emilia, Italy
franco.zambonelli@unimore.it

Abstract Given the growth of agent-based models and technologies in the last decade, nowadays the applicability of agent-oriented techniques to the engineering of complex systems such as pervasive computing ones critically depends on the availability and effectiveness of agent-oriented methodologies. Accordingly, in this paper we take SAPERE pervasive service ecosystems as a reference, and introduce a novel agent-oriented approach aimed at engineering SAPERE systems as multi-agent systems.

1 Introduction

The ICT landscape has dramatically changed with the advent of mobile and pervasive computing technologies. The dense spread in our everyday environment of sensor networks, RFID tags, along with the mass diffusion of always-on-line smart phones and mobile social networking, is contributing to shape an integrated infrastructure that can be used for the provisioning of innovative general-purpose digital services [1,2]. In particular, such infrastructure will be used to ubiquitously access services improving interaction with the surrounding physical world as well as the social activities therein. Users will be expectedly able to deploy customised services, making the overall infrastructure as open as the Web currently is [3].

According to the above trends, a great deal of research activity in pervasive computing and service systems has been recently devoted to solve problems associated to the design and development of effective pervasive service systems. They include: supporting self-configuration and context-aware spontaneous composition; enforcing context-awareness and self-adaptability; and ensuring that service frameworks can be highly-adaptive and very long-lasting [4]. Unfortunately, most of the solutions so far are proposed in terms of “add-ons” to be integrated in existing frameworks [5,6,7]. The result is often an increased complexity of current frameworks and, in the end, a lack of clean and usable methodological approaches to the engineering of complex pervasive services systems.

Against this background, here we elaborate on the SAPERE novel approach to the engineering of complex pervasive service system. SAPERE (short for
“Self-Aware PERvasive service Ecosystems”) tackles the problem of engineering distributed pervasive service systems by a foundational re-thinking of distributed systems, i.e., grounding on a nature-inspired [8], and specifically bio-chemically inspired approach, to effectively support context-awareness, spontaneous service composition, and self-adaptivity. Specifically, SAPERE attacks the program of engineering adaptive pervasive service systems by:

- Modelling and architecting a pervasive infrastructure as a non-layered spatial substrate, hosting the execution of an ecosystem of distributed software agents, each associated to the various individual components of the infrastructure—e.g., devices, sensors, or software services.
- Exploiting the spatial substrate as a sort of shared coordination medium [9] for the agents of the ecosystem. Such a substrate embeds the basic coordination laws (eco-laws), which have a bio-chemical inspiration (i.e., agents manifest their activities by data-items acting as sort of chemical molecules that interact by bonding with each other and diffusing across space).
- Making the overall ecosystem behaviour be driven by the spontaneous dynamics resulting from applying the eco-laws, leading to the unplanned, i.e., self-organising, composition of distributed components, and inherently supporting dynamic context-aware and self-adaptive behaviour.

The SAPERE approach makes it easy to develop adaptive pervasive, due to both its rather intuitive programming model and its clean accompanying software engineering methodology.

Accordingly, the remainder of this paper is organised as follows. Section 2 motivates the SAPERE approach and sketches its overall agent-based architecture. Section 3 overviews and exemplifies the underlying programming model along with its coordination model based on eco-laws. Section 4 presents the methodology defined to support the design and development of complex pervasive service systems as multi-agent systems (MAS) based on the SAPERE approach. Section 5 discusses some related work and concludes the paper.

2 MAS for Pervasive Service Ecosystems in SAPERE

SAPERE targets emerging pervasive computing scenarios based on agent-based abstractions. This calls for specific requirements for SAPERE systems (Subsection 2.1), and also leads to a specific agent-oriented meta-model (Subsection 2.2).

2.1 Basic Requirements

The first key requirement is situatedness in the physical and social environment. In SAPERE pervasive systems, each agent represents individuals, software, and data tightly linked to a given space-time situation, which should affect the overall system only based on some notion of locality that can take into account physical issues (such as the position in an articulated environment) or social ones (such
As who triggered some activity, and which are his/her social profile and relationships. Accordingly, the underlying meta-model should make sure that agents can access to (and influence) only a limited portion of the overall environment.

The second key requirement is self-adaptivity. The overall MAS should exhibit the inner ability to intercept relevant distributed situations, even those not explicitly considered at design-time, and accordingly react with no global supervision to achieve the overall system goals—both implicit and explicit ones. This should be achieved by spontaneous re-distribution and re-shaping of the overall system information and activities.

Finally, since emergent pervasive computing scenarios are based on the opportunistic encounter of devices, humans, data and activities, with no prior knowledge of each other, a high degree of openness is required, which should reflect in the use of semantic-based and fully decoupled interaction mechanisms.

2.2 The SAPERE Meta-model

Once the main requirements for SAPERE systems are introduced, the main abstractions of the SAPERE meta-model can be defined, which tailors multi-agent systems (MAS) for pervasive computing scenarios.

**Agents** — Agents are the main abstraction in the SAPERE model. As the loci encapsulating autonomy and control, agents are the natural means to model sensors and actuators of pervasive computing system, as well as software services (i.e., web services, situation recognisers, local monitors), and the software managing handheld devices carried by humans.

**LSA** — Because of the need of coordinating different kinds of entities in an open way and without global supervision, a cornerstone of the SAPERE approach is that agents manifest their existence in the MAS by a uniform representation called a Live Semantic Annotation (LSA). An LSA exposes every information about the agent (state, interface, goal, knowledge) that is pertinent for the system: it is live since it should continuously reflect changes in the agent state; it is semantic since it should be implicitly or explicitly connected to the context in which such information is produced, interpreted and manipulated; and it has the form of an annotation, i.e., a structured piece of information resembling a resource description—as in RDF.

**LSA-space** — Manifestation of LSAs is supported by the so-called LSA-space, acting as the true fabric of all interactions. There, LSAs are injected by agents, float, and evolve, ultimately reifying all the required information about system activities and processes. The LSA-space is distributed among all devices of the pervasive computing system: the portion of the LSA-space that represents a single locality of the environment is called local LSA-space.

**LSA bonding** — In order to make any agent act in a meaningful way with respect to the context in which it is situated, special mechanisms are needed to control the sphere of influence of each agent. To this end, LSAs can include bonds (i.e., references) to other LSAs in the same context. Only via a bond to another LSA an agent can read its information, inspect the state/interface of another agent, and act accordingly.
Eco-laws — Because of adaptivity, while agents enact their *individual* behaviour by observing their context and updating their LSAs, *global* behaviour (i.e., global coordination in the MAS) is enacted by rules manipulating the LSA-space, called *eco-laws*. Eco-laws can perform deletion/update/movement/re-bonding actions applied to a small set of LSAs in the same locality—similarly to how chemical laws affect molecules.

Thus, agents inject LSAs in the space, which by proper diffusion and aggregation eco-laws establish *fields data structures* [10,11,12] of LSAs, which cover subparts of the network and carry information about the originating LSAs (and agent) and its position in the network. Any agent interested in reading such information shall then autonomously manifest this fact in its LSA, which by proper bonding eco-laws will then bond to the local LSA of the field. After all the required information has been read, the agent can affect the field originator by injecting itself an LSA, which spreads back, reach the originator’s side, and is read through the same bonding mechanism.

3 Programming SAPERE Systems: API & Examples

In this section we overview how SAPERE applications can be programmed, by introducing some of the API of the SAPERE middleware and exemplifying its usage. While the whole articulation of SAPERE programming cannot be fully described here, we intend at least to give readers a clue, and also enable them to better understand the overall SAPERE development methodology.

As for any distributed environment, the execution of SAPERE applications is supported by a middleware infrastructure [13]. The infrastructure is lightweight, and enable a SAPERE node to be installed in tablets and smartphones. From the operational point of view, all SAPERE nodes are at the same level since the middleware code they run could support the same services, and provides the same set of functions—i.e., hosting the LSA space and the eco-laws engine.

From the viewpoint of the individual agents constituting the basic execution unit, middleware provides them with an API for advertising themselves via LSAs, and to support LSA continuous updating. In addition, API enables agents to detect local events, such as the change of some LSAs, or, the enactment of some eco-laws on available LSAs. Eco-laws are built as a set of rules embedded in SAPERE nodes, each hosting a local LSA-space. For each node, the same eco-laws apply to rule the dynamics of both local LSAs (in the form of bonding, aggregation, and decay), and non-locally-situated LSAs (via the spreading eco-law that can propagate LSAs through distributed nodes).

From the viewpoint of the underlying network infrastructure, the middleware transparently absorbs dynamic changes at the arrival/dismissing of supporting devices, without affecting the individual perception of the spatial environment.

3.1 The SAPERE API

In the SAPERE model, each agent executing on a node takes care of initialising at least one LSA, and possibly more, of injecting them on the local LSA space,
AgentNoiseSensor {
    init() {
        float nl = sample();
        injectLSA([sensor-type = noise; accuracy = 0.1; noise-level = nl]);
    }
    run() {
        while(true) {
            sleep(100);
            float nl = sample();
            updateLSA(noise-level = nl);
        }
    }
}

Figure 1. Pseudo-code of a noise sensor.

and of keeping the values of such LSAs updated to reflect its current situation. Each agent can modify only its own LSAs, and eventually read the LSAs to which has been linked to by a proper bonding eco-law. Moreover LSAs can be manipulated by eco-laws, as explained in the following sections.

The SAPERE middleware provides agents with the following API:

- `injectLSA(lsa)` is used by agents to inject an LSA into the tuple space. Each agents must inject at least one LSA at initialisation to exist within the SAPERE ecosystem.
- `updateLSA(field, new-value)` make agents atomically update some fields of an LSA to keep it alive. The idea is that specific threads inside an agents are launched to ensure that the values of LSAs that are to be kept alive are promptly updated.
- A set of `onEcoLawEvent(lsa)` methods makes it possible for an agent to sense and handle whatever events occur on its LSAs. For example, the `onBond(lsa)` method allows the event represented by the LSA to be bond with another LSA matching the former.

As a first example, Figure 1 reports the (pseudo-)code of an agent that acts as a noise sensor, injecting an LSA with noise level, and periodically updating it.

3.2 Matching and Bonding

More generally, LSAs are built as descriptive tuples made by a number of fields in the form of “name-value” properties, and possibly organised in a hierarchical way: the value of a property can be a SubDescription—a set of “name-value” properties, again. By building over tuple-based models [14], the values in a LSA can be either actual – yet possibly dynamic and changing over time (which makes LSAs live), or formal, that is, not tied to any actual value unless bond to one and representing a dangling connection (typically represented with a “?”).

Pattern matching between LSAs – which is at the basis of the triggering of eco-laws – happens when all the properties of a description match, i.e., when
Agent AccessNoiseInformation {
    init() {
        injectLSA(sensor-type = noise; noise-level = "?");
    }
    onBond(LSA b) {
        float nl = b.noise-level();
        print("current level of noise = "+ nl);
    }
}

Figure 2. An agent that inject an LSA matching with that of the noise sensor and enables it to access the corresponding noise-level information.

for each property whose names correspond (i.e., are semantically equivalent) the associated values match. As in classical tuple-based approaches, a formal value matches with any corresponding actual value [9]. For instance, the LSA of the noise sensor in Figure 1 can match the following (sensor-type = noise; noise-level = ?), expressing a request for acquiring the current noise level.

The properties in the first LSA (e.g., accuracy) are not taken into account by the matching function which considers only inclusive match. The basic reaction of the LSA-space in the presence of two matching LSAs is to bond them.

Bonding upon match is the primary form of interaction among co-located agents in SAPERE—i.e., within the same LSA-space. In particular, bonding can be used to locally discover and access information, as well as to get in touch with and access local services—all of which with a single and unique adaptive mechanism. Basically, the bonding eco-law implements a sort of a virtual link between LSAs, whenever two LSAs (or some SubDescriptions within) match, by connecting the respective formal and actual values in a sort of bidirectional and symmetric link: the two agents holding bond LSAs can read each other’s LSAs, thus enabling exchange of information.

Thus, once a formal value of an LSA matches with an actual value in an LSA it is bound to, the corresponding agent can access the actual values associated with the formal ones. For instance, the AccessNoiseInformation agent in Figure 2 injects an LSA matching that of Figure 1, thus enabling AgentNoiseSensor in Figure 1 to access the corresponding noise level information.

Bonding is automatically triggered upon match—that is, the middleware looks for possible bonding upon any relevant change to the LSAs. Analogously, de-bonding takes place automatically whenever matching conditions no longer hold due to some changes to the actual “live” values of some LSAs.

3.3 From Bonding to Service Composition

The above example shows how to program SAPERE agents and, depending on the LSAs injected by such agents, how bonding takes place along with exchange of information. However, it is also possible to express a formal field with the syntax “!” to represent a field that is formal unless the other “?” field has been
bond. This makes it possible for an LSA to express parameterised services, where “?" represents the parameter of the service, and “!" field represents the answer that it is able to provide once it has been filled with the parameters.

It should be noted that the bonding eco-law mechanism can be used to enable two agents to spontaneously get in touch with each other and exchange information with a single operation—and, in the case of “!”, automatically composing two components and have the first one automatically invoking the services of the second one. That is, unlike traditional discovery of data and services, bonding makes it possible to compose services without distinguishing between the roles of the involved agents and subsuming the traditionally-separated phases of discovery and invocation.

3.4 Aggregation, Decay, and Spreading

The additional eco-laws of aggregation, spreading, and decay can be triggered by agents simply by injecting LSAs with specific properties.

The aggregation eco-law means to aggregate LSAs together so as to compute summaries of the current system context. An agent can inject an LSA with an aggregate and type properties. The aggregate property identifies a function to base the aggregation upon. The type property identifies which LSAs to aggregate. In particular, it identifies a numerical property of LSAs to be aggregated. In the current implementation, the aggregation eco-law is capable of performing most common order and duplicate insensitive (ODI) aggregation functions [15,16].

The decay eco-law enables the vanishing of components from the SAPERE environment: it applies to all LSAs that specify a decay property to update the remaining time to live according to the specific decay function, or actually removing LSAs that, based on their decay property, are expired. For instance, [sensor-type = noise; noise-level = 10; DECAY=1000], makes LSAs be automatically deleted after a second.

The spreading eco-law – unlike the two above that act on a single LSA space – enable non-local interactions, and specifically provides a mechanism to send information to remote LSA spaces, and make it possible to distribute information and results across LSA spaces. One of the primary usages of the spreading eco-law is to enable searches for components that are not available locally, and vice versa to enable the remote advertisement of services. For an LSA to be subject to the spread eco-law, it has to include a diffusion field, whose value (along with additional parameters) defines the specific type of propagation.

3.5 Towards Self-organisation Patterns

The eco-laws described above represent a necessary and complete to effectively support self-organising, nature-inspired interactions. In fact, by shaping LSAs so as to properly trigger eco-laws in a combined way, it is possible to realise a variety of self-adaptive and self-organising patterns.
For example, aggregation applied to the multiple copies of diffused LSAs can reduce the number of redundant LSAs so as to form a distributed gradient structure, also known as computational force fields [17]. As detailed in [18,10,11], many different classes of self-organised motion coordination schemes, self-assembly, and distributed navigation can be expressed in terms of gradients. By bringing also the decay eco-law into play, it is possible to build pheromone-based distributed data structures. Further examples can be found in [12].

4 Engineering SAPERE Systems: The Methodology

According to Osterweil [19] “software processes are software too”: so, in order to build the SAPERE methodology, we followed a path that corresponds to the design of a software system. Thus, we first define the set of the SAPERE methodology requirements (Subsection 4.1); then we design the SAPERE methodology process (Subsection 4.2).

4.1 Requirements for the SAPERE Methodology

The first, obvious requirement is that the SAPERE methodology should support the design and development of SAPERE pervasive service ecosystems according to the above-mentioned meta-model. From an analysis of the state of art in the Software Engineering area [20] the following methodology requirements can be pointed out:

– Due to the nature of the application domain, the more appealing process model is seemingly the iterative model, allowing engineers to iterate the different phases in order to obtain the best design.
– The SAPERE process could be organised in five main phases (Requirements Analysis, Analysis, Architectural Design, Detailed Design, and Implementation) in order to maintain the coherence with the general structure of standard design methodologies. This should facilitate the understanding of the methodology also for non-domain experts.
– The first two phases (Requirements Analysis, Analysis) should be very similar to the traditional analysis phases. On the one hand, this should make the adoption of the methodology easier to non-domain experts; on the other hand, it is generally understood that the analysis phase investigates the so-called “problem domain”, and the “problem” is not directly related to the technologies adopted for resolving it.
– The methodology should provide specific activities supporting the designer in the choice of architectural patterns and self-* mechanisms, in order to address the modelling of coordination and services. Coordination should be considered as an emergent property, so that a specific self-organising pattern could be chosen in order to obtain the required coordination goal.
– Since SAPERE deals with the investigation of self-aware pervasive ecosystems, the SAPERE methodology should deal with specific activities of simulation and verification in the Architectural Design phase. In particular,
simulation should take inspiration from the existing related works such as
[21,22], where a suite of activities such as “Exact Verification”, “Simulation”,
and “Tuning” are already defined in a method fragment. However, the
SAPERE methodology should not adopt the proposed fragment as is, but
should instead provide a specific version of the aforementioned activities—
namely, “Exact Prediction”, “Approximate Prediction”, “Simulation”, and
“Tuning”. Also, the methodology should provide specific activities for “Veri-
fication” and “Quantitative Measures” (respectively, in the Detailed Design
and in the Implementation phases) which could provide engineers with ef-
fective data and information about the behaviour of the running system.

- From the meta-model point of view, taking inspiration from the work done
in the AOSE field [20], the methodology meta-model should be created ac-
cording to the transformational structure – i.e., each phase/domain should
feature its own set of abstractions as in Model-Driven Engineering – for the
sake of clarity, and to make it easier to move from one phase to another.
- The meta-model abstractions belonging to the Requirements Analysis and
Analysis phases should come both from traditional problem analysis and
from some AOSE methodologies where environment abstractions and envir-
onment topology are first-class abstractions. This allows the environment to
be taken into account since the first phases of the process.
- The meta-model abstractions belonging to Architectural Design and De-
tailed Design should be created ex-novo drawing from the SAPERE meta-
model described in Section 2. In particular, the work done in [23] about the
chemical metaphor could be very useful for the identification of the design
abstractions.

4.2 The SAPERE Process

The SAPERE methodology is illustrated following the IEEE-FIPA Standard
Design Process Documentation Template (DPDT) [24], developed to facilitate
understanding of the methodology (as an internationally recognised standard),
as well as the comparison with others. For the sake of brevity, in the following
we shall only outline the main features of the SAPERE methodology.

The Lifecycle The SAPERE methodology lifecycle is an iterative process
composed by five main phases: Requirements Analysis, Analysis, Architectural
Design, Detailed Design, and Implementation (Figure 3).

The Meta-model The meta-model of the SAPERE methodology is reported in
Figure 4. On the one hand, it complies with the transformational structure (see
Subsection 4.1); on the other hand, it is organised in four different domains re-
flecting the first four methodology phases. Regarding the Implementation phase,
a specific meta-model is not required here since the design abstractions have to
be mapped on to the SAPERE middleware abstractions. Here we only report the
ideas that inspired the meta-model construction. In particular, the abstractions
of the Detailed Design phase come from the SAPERE abstract model, whereas the abstractions of the Architectural Design have many sources: (i) the SAPERE abstract model – Annotation, Manifest, Context, Behaviour, Place, Topology –, (ii) the self-organisation domain – SelfOrganising Pattern, and SelfOrganising Mechanism –, and (iii) the AOSE methodologies—Role.

For the abstractions of the Analysis phases we take inspiration from the main AOSE methodologies [20]. In particular, for the environmental and interaction aspects we adopt SODA style, since the SODA methodology [25] specifically focuses on the modelling and design of both environment and interaction [26]. Environment modelling starts since the Requirements Analysis phases (Legacy Environment), then during the Analysis phase we derive the services (Service) from both the system requirements (Requirement) identified in the previous phase, and from legacy resources. Also, the environment topology is modelled since the Analysis phase (Virtual Topology).

Interaction issues are captured in the Requirements Analysis by the Relation concept, which represents any kind of relationships among requirements, and between requirement and legacy environment. In the Analysis phase, the Relation generates – red arrow in Figure 4 – both Interaction and Constraint. Interactions represent the acts of interaction among Tasks, among Services and between Tasks and Services; Constraints, instead, enable and bound the entities’ behaviour.

Finally, in order to correctly model the requirements, in the Analysis phase we decided to perform first a goal-oriented analysis (Goal), then to derive tasks (Task) by goals—as done in [27].

The Phases Here we introduce the five SAPERE methodology phases, by shortly discussing the high level process diagrams.

Figure 5(left) presents the process diagram of the Requirements Analysis phase, composed by three main activities, namely Requirements Modelling, Legacy Environment Modelling, Relations Modelling. There, requirements, legacy resources and relations and dependencies among them are analysed. In this phase,
traditional techniques coming from the AOSE field are adopted for analysing both the requirements and the legacy environment.

Figure 5 (right) presents the process diagram of the Analysis phase. The Analysis is composed by five main activities. In particular, Goals Analysis and Task Analysis lead the engineers to identify firstly the system’s goals and then the tasks necessary to accomplish them. Services Analysis is devoted to derive and to analyse the system’s services coming both from the legacy environment and from the system’s requirements, while Virtual Topology Analysis analyses the system’s environment topology. Finally, Interactions Analysis and Constraints Analysis analyse respectively the interactions among system’s entities and the possible constraints about entities behaviours or about system’s environment.
Figure 5. Requirements Analysis (left) and Analysis (right) activities diagrams

Figure 6 presents the process diagram of the Architectural Design phase. This phase is composed by eight main activities, namely Topologies Design,
SelfOrganisations Design, Roles Design, Models Extraction, Exact Prediction, Approximate Prediction, Simulation, and Tuning. The process here is more complex since now, after problem analysis, the system have to be designed according to the SAPERE philosophy. In particular, the first three activities – Topologies Design, SelfOrganisations Design, Roles Design – define the models for system roles (their behaviours and interactions), the self-organisation mechanisms realising the services identified in the analysis, and the topological structure of the environment. Then, taking inspiration from [21,22], we design five activities (Models Extraction, Exact Prediction, Approximate Prediction, Simulation, and Tuning) devoted to system prediction and simulation. Thus, the effect of the architectural design on the system behaviour could be verified through the study of emerging properties. Adopting simulation during architectural design makes it possible for engineers the early discovery of problems due to either unsatisfactory architectural choice or inaccurate problem analysis.

Figure 7(left) presents the process diagram of the Detailed Design phase. This phase is composed by five main activities, namely Eco-Laws Design, Agents Design, Neighborhood Design, Bonds Design, Verification. The first four activities are devoted to the detailed design of system according to the SAPERE abstract model, while Verification allows engineers to effectively verify the behaviour of the whole system entities before starting the implementation phase.

Finally, Figure 7(right) presents the process diagram of the Implementation phase. This phase is composed by four main activities namely Middleware Adaptation, Coding, Testing, Quantitative Measures. Middleware Adaptation plays a key role in this phase since in this activity the detailed design entities have to be mapped onto the middleware entities. This activity should be “trivial” –
i.e., one-to-one mapping – if the middleware totally supports the detailed design entities, otherwise it could be very complex and require a lot of re-engineering work such as the ex-novo creation of ad hoc self-organisation mechanisms. Then, *Coding* has to start before *Testing*, but after that their executions could be interleaved. Only when the system developing is concluded it is possible to execute specific *Quantitative Measures* activity – for testing the system requirements satisfaction accuracy.

## 5 Final Discussion

The definition of agent-specific methodologies is definitely one of the most explored topics in Agent-Oriented Software Engineering (AOSE): a large number of AOSE methodologies describing how the process of building a MAS should/could be organised has been proposed in the literature. However, what characterises most of the methodologies proposed so far is that they assume a very traditional process models [28] – from analysis to design, implementation, and maintenance – for organising the process of MAS engineering.

It appears rather odd that most proposals for AO methodologies adopt a standard process model when, in the real world of industrial software development, such a standard model is rarely applied. It is a matter of fact that, in many cases, software is developed following a non-structured process: analysis, design, and implementation, often collapse into the frenetic work of a bunch of technicians and programmers, directly interacting with clients (to refine typically vague specifications), and striving to deliver the work on time [29]. In the mainstream community of Software Engineering such a situation is getting properly attributed via the definition of novel software process models, specifically conceived to give some flavour of “engineering” to such chaotic and frenetic processes—e.g., agile and extreme software process models, and method engineering [30,31]. In the area of AOSE, a similar direction should be explored too, possibly exploiting the fact that the very abstractions of agents may promote the identification of different and more agile process models and method engineering [32]. For a rather exhaustive survey of all the related activities in the AOSE field, we refer the interested reader to [20].

The definition of a proper process for the engineering of SAPERE pervasive service ecosystems was one of the main motivations behind this work. In order to allow the reader to fully understand the SAPERE process, in this paper we first introduce the SAPERE model, then discuss how a SAPERE system could be programmed, providing some simple examples, finally we illustrate the SAPERE methodology, by defining the software development process. While the space for this paper is not enough to provide the reader all the details of the SAPERE methodology, here we discuss the main issues of the engineering of pervasive service ecosystems according to the SAPERE approach, thus showing how agent-oriented technologies and methodologies can be effective in the design and development of complex software systems.
Acknowledgements

This work has been supported by the EU-FP7-FET Proactive project SAPERE – Self-Aware PERvasive service Ecosystems, under contract no. 256873.

References


Engineering Industrial Multi-Agent Systems
The JIAC V Approach

Marco Lützenberger, Tobias Küster, Thomas Konnerth, Alexander Thiele, Nils Masuch, Axel Heßler, Jan Keiser, Michael Burkhardt, Silvan Kaiser, Jakob Tonn, and Sahin Albayrak
firstname.lastname@dai-labor.de
DAI-Labor, Technische Universität Berlin

Abstract. The community of agent researchers and engineers has produced a number of interesting and mature results. However, agent technology is still not widely adopted by industrial software developers or software companies. Yet, given that the software paradigms which are currently employed by the software industry, such as service-oriented architectures or cloud computing, have much in common with agent-oriented software engineering, industrial software projects could greatly benefit from agent technology. In this paper, we make an approach to analyse the requirements of current industry-driven software projects and show how we were able to cope with these requirements in the JIAC V agent framework. We argue that the lack of industry-grade requirements and features in other agent frameworks is one of the reasons for the slow acceptance of agent technology in the software industry. The JIAC V framework tries to bridge that gap — not as a final solution, but as a stepping stone towards industrial acceptance.

1 Introduction

The concept of Agent Oriented Software Engineering, or AOSE, dates back as far as 1997, when Michael Wooldridge published his widely cited and influential article [34] and established an entirely new branch of research. Today—fifteen years later—there are countless theories, methodologies, tools and frameworks, each supporting the development of agent-based software applications in the one or the other aspect. Yet, despite the multitude of scientific works on AOSE, it is far from being audacious to say that research in agent technology has as yet failed to convince the industry to adopt their ideas [1]. The reasons for this are not entirely clear, but it is obvious that certain incentives are lacking [33].

In this paper we present the fifth iteration of the Java Intelligent Agent Componentware framework (JIAC) [14]. We are well aware that after fifteen years of research, the enthusiasm for ‘yet another agent framework’ can only be moderate. Yet, JIAC does not fall into this category. We argue that—as opposed to well known and established agent frameworks—JIAC was neither explicitly developed as a research framework (cf. Jason [5]), nor streamlined towards the
requirements of individual industrial stakeholders (cf. JADE [3]). JIAC was developed under premise to cover a wide spectrum of requirements and to further the industrial adoption process. Originated in the year 2002 and being in its fifth incarnation today, JIAC has reached technical maturity. The development was focused on the objective to provide a robust communication infrastructure. We implemented this infrastructure based on ActiveMQ and thus allow for a reliable inter-agent communication, even beyond the borders of homogeneous computer networks. Additional agent capabilities are encapsulated within reusable modules, namely AgentBeans which can be added as needed. This mechanism allows for multi-agent system (MAS) solutions which are tailored to the application context. Following this mechanism, JIAC currently offers modules for migration, rule interpretation, persistence, scripting languages, load measurement, OSGi-integration and human-agent communication, to name but a few.

Despite many practical appliances with convincing results, our experience is that JIAC is only marginally known within the agent community. As such, it is the aim of this paper to present the JIAC framework with particular emphasis on features that we regard to be beneficial for industrial projects. We do not consider JIAC to be an ultimate solution, but as a step towards industrial acceptance. We argue that, so far, state-of-the-art frameworks were not able to convince the industry of the elegance of the agent paradigm and alternative approaches are strongly required, should AOSE ever gain foothold in industrial processes.

We begin this paper with a brief description of different projects in which JIAC was used and respectively mention features that were required for a successful appliance (Section 2). Based on this analysis, we examine the capabilities of well established agent frameworks to deal with the previously collected requirements (Section 3). We use this analysis to substantiate our thesis, that certain features are not sufficiently covered in state-of-the-art approaches. We proceed by presenting the JIAC framework in more detail (Section 4), including a description of standard features, the most relevant extensions as well as development tools. Subsequently, we describe selected appliances of JIAC in more detail and respectively underline the technical integration of required framework features (Section 5). Finally, we discuss the role of JIAC within the pool of well established agent frameworks and the agent community and wrap up with a conclusion (Section 6).

2 The Case of the JIAC V Framework

In this section we compile a list of requirements that we derived from our application projects over the last couple of years. In order to achieve this, we first give a short description of the projects in which the JIAC V agent framework was used and developed. We further emphasise the different domains in which the framework was applied. We conclude this section with an explanation of the requirements that arose from these very applications and guided the development of JIAC V.
2.1 Project Summaries

The goal of the Service Centric Home (SerCHo) project was the development of an open service platform that increases life quality at home. The platform was intended to support the quick and easy delivery of new context sensitive services into the home environment and the provisioning of a consistent user interface for these services. Within the project we have focused on developing a service engineering methodology and tool suite as well as a service delivery platform to simplify service development, deployment and maintenance.

The technologies developed in the Multi Access – Modular Services (MAMS+) project allow non-technical persons to fast and easily create, deploy and manage services, according to the users’ needs. We have developed a service delivery platform based on our multi-agent framework [30]. The platform integrates modern technologies like IMS/SIP, allows for service composition and features service matching, load-balancing and self-healing mechanisms, to name but a few.

Within the project Gesteuertes Laden V2.0 (GL V2.0, Managed Charging V2.0) [32] the goal was to develop a decentralised intelligent energy management system that utilises electric vehicle’s batteries as mobile energy storages. The purpose of the developed planning algorithms was to stabilise the energy grid and to maximise the amount of renewable energy within the electric vehicles (EVs), while taking into account forecasts of available wind energy.

In order to maintain a good standard of living for senior citizens, new technologies have been developed within the SmartSenior project. Our work included sensor-based situation detection and reaction as well as notification and remote management [29]. During a field study the solutions have been successfully installed and tested in the home environments of more than thirty elderly participants.

The project Energy Efficiency Controlling in the Automotive Industry, or EnEffCo, aims at the implementation of a modular software system [19] to simulate operational modes of plant sections with relevant energy consumption. The software serves as a tool for decision makers in manufacturing, to whom it offers the identification and evaluation of strategies and tactics for establishing cost- and energy-efficient production schedules.

Intelligent Solutions for Protecting Interdependent Critical Infrastructures (ILIas) is a project aimed towards developing intelligent solutions for protecting critical infrastructures that provide electricity and telecommunication services to the general public [18]. These solutions need to be scalable and reconcile the need for fast automated reaction with manual supervision for highly critical decisions. Software solutions and protection mechanism efficiencies in large-scale networks are evaluated using simulated disaster scenarios. The simulation models are supplemented by a hardware test laboratory where exemplary interdependent energy and telecommunication infrastructures are set up.

The on-going project BeMobility 2.0 investigates the integration of electric vehicles (EVs) into urban transport and energy networks. In addition to the development of concepts that will combine different mobility services (e.g. vehicles, public transportation, etc.), an energy management system [9] for a Micro
Smart Grid is being developed, in which a variety of system components, such as EVs, charging infrastructure, and energy sources, are taken into account.

The aim of the Connected Living project is to provide a system for integrating and managing the several ‘smart devices’ in future home environments. Besides providing a layer of abstraction for controlling devices by diverse vendors, another goal is to supply an infrastructure for developing, publishing and deploying ‘assistants’ to the users’ home environments. In this context, an ‘assistant’ is a single agent or a coalition of agents that help a future home user to achieve his goals.

The objective of the project Extensible Architecture and Service Infrastructure for Cloud-aware Software, or EASI Clouds, is to provide a comprehensive and easy-to-use cloud computing infrastructure with support for cloud interoperability and federation. The infrastructure will include advanced SLA (Service Level Agreement) management for all service layers, facilities for capacity planning, heterogeneous provisioning as well as accounting and billing.

The Multi-Agent Programming Contest is an annual competition in agent based artificial intelligence that started in 2005 [2]. The contest is an attempt to stimulate research in the field of programming multi-agent system. Our team participated since 2007. We use it as a platform to teach students in the field of agent-based design and implementation using the JIAC V agent framework.

2.2 Requirements Derived from the Projects

During the process of domain analysis and system design of those projects, several requirements have been identified and hence fulfilled in JIAC V. While many of them are typical for industrial or business software frameworks, it is our believe that a multi-agent framework does not have to stand behind.

In many projects the results had to be tested during field trials or user roll-outs. Applications had to be running for months without problems. Therefore, stability and robustness are key issues for a good user experience. A certain level of robustness was important, especially in dynamic environments, e.g. deploying and un-deploying new services or agents should not affect other parts of the application. The same holds true in a distributed context, e.g. when new nodes join or leave a system or agents migrate between them. The framework had to be able to handle a potentially large number of agents and agent nodes without a decrease in performance, a requirement especially affecting communication infrastructure and the distributed service directory. Several projects dealt with service delivery and management of services, resulting in various requirements like support for service life-cycle, management interfaces, runtime deployment and third-party service integration. Additional requirements related to management and adaptive behaviour are monitoring and introspection. It has to be possible to retrieve status information from all framework components in a standardised way. Certain functionalities were required in multiple projects so that component reuse became necessary. Additionally, the framework needed to be extensible in order to be able to integrate future requirements.
3 State of the Art

When we compared the requirements of our projects to existing agent frameworks, the results were twofold. On the one hand, platforms like Jason [4] or 3APL [12] have been created on a very strong theoretical background. They feature elaborate implementations of cognitive concepts that are important for the agent research agenda as a whole. Even though there have been approaches to extend these frameworks—e.g. JASDL [17] for Jason—they were never intended to be industry-ready but geared towards research. As such, they do not fulfil our requirements when implementing large scale projects.

More pragmatic approaches such as JADE [3] or the JACK [7] framework are more focused on the engineering and development aspects of applications. However, development of the JACK framework seems to have stopped. The JADE framework on the other hand has a long list of extensions and additions, such as the Web Service Integration Gateway [10], AgentOWL [21], WADE [8] or the MASE framework [27]. On a point-by-point basis these extensions seem to fulfil many of our requirements. However, most of these extensions have been developed independently from each other and using them within the same software project will be tedious if not impossible. Thus, while many of the approaches are well thought-out and useful, the JADE framework with its extensions lacks the coherence and unity that we would expect from a modern software framework.

A current agent architecture that tries to bridge the gap between agent technology and the software industry is the Jadex framework [28]. The developers of Jadex have recently taken a number of approaches to improve Jadex in ways that make it more compatible with industry standards [6].

However, while we appreciate the approach to adapt the framework to industry needs, we find that a number of design decisions do not comply with the requirements for our typical projects. The decision to base the framework on the active component model—with agents as the internal architecture of the components—reverses the control architecture from our point of view. We regard agents as the surrounding structure and expect them to have capabilities that enable communication and interaction.

For the above reasons, existing agent frameworks either do not fulfil our requirements for practical applications, or their models are too different from our modelling approach for agent oriented applications. In the following we describe the JIAC framework, which represents our approach to an agent architecture that fulfils our needs.

4 The JIAC V Framework

JIAC V is a Java-based multi-agent development framework and runtime environment [24] that has been both developed and deployed in a number of application projects. Based on the requirements of those projects (see Section 2) particular emphasis has been placed on the following aspects:

- robustness, scalability, modularity and extensibility
– adoption of a service-oriented view and integration of third-party services provided e.g. as web services and/or OSGi bundles
– dynamically adding and removing services, agents, and nodes at runtime
– extensive tool support, both at design time (modelling and development) and at run time (management and monitoring)

In the following, we describe the JIAC framework in detail, and how those requirements are satisfied.

4.1 Core Mechanisms of JIAC Agents

One of the core aspects of JIAC is the integration of agents with the service-oriented architecture paradigm (SOA) [13]. Using a powerful discovery and messaging infrastructure, JIAC agents can be distributed transparently over the network, or even beyond network boundaries. An agent-platform comprises one or more ‘agent nodes’ which are physically distributed and provide the runtime environment for JIAC agents. New agents, services, as well as further agent nodes can be deployed at runtime. Agents can interact with each other by means of service invocation, by sending messages to individual agents or multicast-channels, and by complex interaction protocols. Each individual agent’s knowledge is stored in a tuple-space based memory. Finally, JIAC agents can be remotely monitored and controlled at runtime via the Java Management Extension Standard (JMX). For an overview of how those individual features were used in projects, refer to Section 5 and Table 1 in Section 6.

Each agent contains a number of default components, such as an execution-cycle, a local memory and the communication adaptors. The agents’ behaviours and capabilities are implemented in a number of so-called AgentBeans. AgentBeans support very flexible activation schemes: A bean may be executed at regular intervals or according to a life-cycle change, such as initialised, or started. Further, AgentBeans can attach observers to the agent’s memory, being called for instance each time the agent receives a message or updates its world model. AgentBeans also provide action methods, which are exposed to the directory and invoked either within the agent or by other agents.

Using these four mechanisms, it is possible to define all of the agents’ capabilities and behaviours [15]. Furthermore, the structure of each agent contains a number of standard components, such as an execution-cycle, a local memory and the communication adaptors. The entire multi-agent system, i.e. which agent has which agent beans and how those agents are distributed over the agent nodes, is then set up using one or more Spring\(^1\) configuration files.

4.2 Default and Extension Components

JIAC agents contain a number of individual AgentBeans that are implemented as described above as well as a set of standard AgentBeans that constitute the

\(^1\) Spring: [http://www.springsource.org/](http://www.springsource.org/)
basic interior of an agent. One such AgentBean each JIAC agent is equipped with by default is the Communication Bean. First, this component manages the inter-agent service communication; second, it allows the agents to exchange messages with other agents or groups of agents on the network, addressing individual agents or multi-casting to message channels. The messages are not restricted to FIPA messages but can have any data as payload.

Complementary to the AgentBeans, there are NodeBeans, adding functionality to the node as a whole. Each agent node is by default equipped with a Directory NodeBean, listing the actions of the different agents, and a Message Broker NodeBean, being the counterpart to the agent’s communication bean and allowing them to transparently send messages from node to node using ActiveMQ\(^2\).

Other commonly used AgentBeans and NodeBeans can be added to a multi-agent system by simply adding the bean to the agent’s configuration. For the composition of services, JIAC includes an Interpreter AgentBean for the execution of the high-level service-oriented scripting language JADL++ \cite{14}. Reactive behaviour of agents can be enabled with a Drools\(^3\) rule engine that can be synchronised with the agents’ memory.

Extensions to the capabilities of nodes and agents include a Migration NodeBean, that enables strong agent migration between agent nodes, a Persistence NodeBean that saves the node configuration and allows for restarting the node later on, and NodeBeans for Load Measurement and Load Balancing that provide cross-node load information and distribute agents over nodes at start- and run-time. In order to support application development, JIAC also provides generic functionalities such as AgentBeans for User Management, Human Agent Interfaces, a Webserver NodeBean running an embedded Jetty-server, and a Web Service Gateway AgentBean that exposes JIAC actions as web services and vice versa. Last but not least, the OSGi Gateway allows JIAC nodes to be executed within an OSGi framework and to access other OSGi services.

### 4.3 Development Methods and Tools

Since JIAC is a Java-based agent framework, the bulk of the development work can be done using conventional Java development tools, such as Eclipse and Maven, as well as supportive tools like XML editors. Still, to improve the efficiency in application development, some additional tools are provided, all of which are integrated directly into the Eclipse IDE.

A special JIAC Project Wizard helps with creating new JIAC projects by generating a uniform project structure, including a readily configured Maven pom.xml file listing the required dependencies and a starter class for running the new JIAC application. Further, several Eclipse views provide information about nodes currently running on the network and the agents and services they

---


\(^3\) JBoss Drools: [http://www.jboss.org/drools/](http://www.jboss.org/drools/)
contain, as well as the possibility to start and to interact with newly created agents and services.

Complementary to those basic development tools and utilities, JIAC agents can be modelled using two high-level graphical editors: The Visual Service Design Tool (VSDT) and the Agent World Editor (AWE). Using the VSDT [20], both the workflows of individual agents as well as their interactions can be modelled as a series of BPMN [26] diagrams. Based on those diagrams, executable JIAC AgentBeans or JADL++ services can be generated. The AWE [23] can be used to create visual representations of entire multi-agent-systems and its components, showing the different agents and agent nodes in a distributed system and the individual services and AgentBeans they provide. From these visual models, the tool can generate the corresponding Spring configuration files and JIAC AgentBean stubs for those systems.

Finally, the running multi-agent system can be monitored and manipulated using the ASGARD agent runtime monitor [31], providing an intuitive, three-dimensional view of all agent nodes and agents running in the local network and the communication between them.

5 Featured Projects

Many features and components shown in the last section were developed as a consequence of project requirement analyses. The resulting modular structure of JIAC enables developers to select tailored functionalities. In the following, we present industry projects from different domains—namely energy, electromobility and health. In doing so we put a focus on system engineering aspects.

5.1 ILIas

The main motivation behind the ILIas project is that modern infrastructures are interdependent, such as power and telecommunication grids. In case of failures, this can create cascading effects in several or all of the involved infrastructures. The objective of the project is to research and create intelligent and scalable management systems that provide prediction and reaction to cascading failure effects, so that actions to stabilise the managed infrastructure can be taken. An example for this is the reaction to power outages and the consequent failure of telecommunication networks in the affected areas.

The approach chosen in ILIas is an agent-based decentralised smart grid management system, which observes and controls the grid. Prediction of grid behaviour is supplied by a simulation of power and telecommunication networks. The management agents are able to interact with both physical as well as simulated smart grid entities, which allows for easy testing of even large scale systems. This simulation is also agent-based and implemented using the NeSSi² simulation framework [11]. The simulation is able to work both, offline, simulating a pre-defined scenario, as well as online, by using the current grid state as a starting point to calculate predictions. The human-machine interface is provided by a visual monitoring application (Figure 1) that visualises the smart grid topology.
When developing the p2p based approach in ILIas the most helpful characteristics of the JIAC V framework where the highly variable communication mechanisms, as well as the general agent based models used. The former where able to automatically adopt to changes in the infrastructure, e.g. when handling failure scenarios, the latter allowed very easy and quickly reconfigurable mappings of a given grid topology into a running management system. The resulting system proved to be very reliable with regard to the overall system stability.

Area of improvement was found in the lack of supported database services. Occasionally larger amounts of data had to be handled, which required database integration in the application development. This functionality could be handled by the framework in the future.

5.2 Gesteuertes Laden V2.0

The goal of Gesteuertes Laden V2.0 [32] was to use electric vehicles as mobile and distributed energy storages in order to utilise the potential of fluctuating wind energy and to stabilise the energy grids. As the driver’s inherent needs for mobility are always the main objective to fulfil, a mechanism was needed that supports the users in planning their charging and potential discharging events without limiting them in their flexibility. The solution to this is implemented as a live system including real EVs (three Mini-E vehicles provided by the project partner BMW) and charging stations.
In the project, a distributed mobility and energy management system was designed in which each of the involved actors—such as driver, vehicle manufacturer, energy provider, charging station, and grid operator—is represented by a software agent [16]. The main contribution of the system is the creation of user-centric day schedules containing journeys, charging and discharging events [25]. In this context many different actor-dependent preferences and constraints had to be considered, such as the driver’s appointments (necessary for deduction of mobility needs), wind forecast, characteristics and current state of the EV, characteristics and availability of charging stations, and energy grid constraints.

The developed system features a high degree of complexity containing eight software agents in the back-end and three agents within each of the EVs (see Figure 2). Altogether more than 100 services are running simultaneously, offering different tasks, ranging from simple information services to complex planning algorithms. For each user and each electric vehicle an additional agent representation is running in the back-end, taking the main responsibility for developing user and vehicle schedules.

The data exchange between EV and back-end agents is based on unreliable telecommunication networks (e.g. UMTS), therefore failover mechanisms are used, ensuring a reconnection after network stabilisation. The coordination of charging and discharging events is processed by the EV agents interacting with the charging stations via power line communication. In order to get all relevant information for the planning procedure, third party services such as wind forecasts and charging station status information were embedded into JIAC via the Web Service Gateway. Further a generic database agent has been developed providing access to MySQL databases. As the user interaction plays a very im-

---

**Fig. 2.** GL V2.0 System Architecture.
important role in this scenario, a smartphone application has been developed. The integration into JIAC was done with the Human Agent Interface. The system was evaluated within a three-week field test [22]. During this time, the need for a monitoring component, which notifies the developers about the services’ availability, became apparent and was subsequently installed. Furthermore, the field test revealed, that the service advertisement messages between the CarPC Nodes and Backend Nodes aggregated to a significant amount of traffic, which became too high for a traffic limited connection, such as UMTS. As a consequence, the service advertisement interval has been increased.

5.3 SmartSenior

An ageing society causes high costs for health care and services which can be addressed by modern IT. Especially elderly people can regain a higher level of quality of life when using such IT, since dedicated health care will only be required on few occasions. Such a system was developed in the SmartSenior project, where the focus was to cover most aspects of daily needs while keeping the system usable. The system uses sensors, processors and effectors in order to detect situations at home for performing appropriate (re-)actions [29]. The system as described below has been installed and tested during a field study in 32 apartments. During the eight weeks of the study the system was running stable with no significant errors.

In each apartment two JIAC nodes containing several agents were running. Both nodes were wrapped as OSGi-Bundles and installed in an OSGi-execution environment (Knopflerfish 3.1). The system at the home side contained the following agents: OSGi-Proxy Agent, WebServer Agent, Sensor Agent, SensorDetection Agent, UPnPSensor Agent, Detection Agent, Reaction Agent, Database Agent, ReactionDeployment Agent, Notification Agent. Each agent was designed to perform a specific task. The interaction among these agents resolved into a global system behaviour. The interaction heavily used JIAC’s communication mechanisms, service invocations as well as interpreter services provided by the ReactionDeployment Agent. While it is beyond the scope of this work to explain each agent in detail, an example will demonstrate the interaction and the features of the JIAC framework that have been used.

The main task of the system is to detect specific situations and react in a predefined manner. A situation is defined as a set of specific sensor values in a certain time frame. Sensors send their information to a gateway and eventually the SensorDetection Agent receives these values and checks whether they are from the right home. Sensor values are send via internal communication to the Sensor Agent, which enriches the original information with additional semantic data, such as sensor type, time, room type and apartment id. This information is send to both the Database and Detection Agent, again using internal communication. The Database Agent includes a hibernate wrapper and stores sensor information in a local MySQL Database. It also provides actions to read from the database. These are later used by the WebServer Agent to display a
histogram of sensor values to the user. The Detection Agent stores the same sensory information in its knowledge base. It contains a Drools rule engine which is used to detect different situations. Each detected situation is passed to the ReactionDeployment Agent, which eventually triggers certain reactions, which in turn are provided by the Reaction Agent. First, the mapping between situations and reactions is modelled in BPMN [26] using the Visual Service Design Tool (VSDT, see Section 4.3). The model includes a trigger for reactions and a set of services to be executed. The mapping model is transformed into a JADL++ script [13] and deployed at runtime into the ReactionDeployment Agent using the distributed service directory and JIAC’s JMX interface (see below). Second, the ReactionDeployment Agent executes the right JADL++ script according to the detected situation by using the JADL++ interpreter agent bean. During script execution the previously specified reaction services are invoked. Finally, the Reaction Agent executes these services, e.g. sending notifications to the user. These are send via the Notification Agent, which is also able to receive notification request from other non-JIAC system components via OSGi-Events using the OSGi-Proxy Agent.

Fig. 3. SmartSenior global architecture and deployment chain.

In order to be able to remotely deploy (and delete) reaction scripts, each sensor node (see Figure 3) connects to the back-end node. The connection needs to be static but fail-safe, therefore it is configured to reconnect every time a disconnect occurs. The back-end node functions as a gateway and provides services to retrieve all registered sensor nodes or to deploy and un-deploy reaction-scripts to a specific sensor node. The VSDT connects to the back-end, allowing for use of the gateway services. Via several Eclipse views a user can choose which services are to be deployed or un-deployed into which apartment.
6 Conclusion

In this paper we presented the JIAC V agent framework. The basic idea behind JIAC was to provide an agent framework which meets industrial requirements and is able to facilitate the industrial adoption of the agent paradigm. As such, we initiated the paper with a brief description of industrial projects in which JIAC was used, and emphasised the features that were required for the technical realisation of each project. After comparing the collected requirements to the capabilities of state-of-the-art framework solutions, we presented JIAC V in more detail, describing the architecture of JIAC multi-agent systems as well as the modular assembly of JIAC agents. We further mentioned basic and extending capabilities of JIAC agents and nodes. Based on this description we elaborated on selected projects (namely ILIas, Gesteuertes Laden V2.0 and SmartSenior) and emphasised the respective technical integration of required framework features.

A comprehensive overview of implemented features as well as their appliances within industrial projects is given in Table 1.

<table>
<thead>
<tr>
<th>Core</th>
<th>SerCHo</th>
<th>MAMS+</th>
<th>GL 2.0</th>
<th>Smart Senior</th>
<th>EnEffCo</th>
<th>ILIas</th>
<th>BeMobility</th>
<th>CL-OS</th>
<th>EASI Clouds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Distribution</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Core</td>
<td>Node Deployment</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Core</td>
<td>Services, SOA</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Core</td>
<td>Message-based Communication</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Core</td>
<td>Service Deployment</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Core</td>
<td>Agent Deployment</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Core</td>
<td>Management and Monitoring</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Plugins</td>
<td>Webserver</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Plugins</td>
<td>Interpreter</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>Rule Engine</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>Migration</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>Persistence</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Plugins</td>
<td>Load Balancing</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>Webservice Gateway</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>OSGi Gateway</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
</tr>
<tr>
<td>Plugins</td>
<td>Human Agent Interface</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Plugins</td>
<td>User Management</td>
<td>-</td>
<td>x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

As mentioned at the beginning, we certainly recognise that there are many agent frameworks available – each one with a focus on particular multi-agent system characteristics. Yet, as of today, the agent community was not able to
convince industrial players to adopt their ideas. As opposed to comparable frameworks, JIAC was never intended to include the cutting edge of agent research but to constitute a robust, reliable, homogeneous and well-documented foundation for the development of agent-based software applications.

It was also our intention to equip JIAC with features which are generally required for extensive industrial appliances. Today, the JIAC framework provides a set of well-evaluated and useful capabilities. Common requirements (such as distribution or access to SOA-compliant services) were integrated as core functionalities. Other, less broadly used features were developed as optional modules.

We do not consider JIAC to be an ultimate solution for the discrepancy between agent research and the applying industry. Yet, given the fact that JIAC was originally streamlined towards industrial projects and also towards ease of use, it is our opinion that JIAC has the potential to provide new incentives for industrial stakeholders and users which are not all too familiar with the agent paradigm, to consider agent technology.

References


32. Vattenfall, BMW, TU Berlin, TU Chemnitz, TU Ilmenau: Increasing the effectiveness and efficiency of the applications wind-to-vehicle (W2V) and vehicle-to-grid (V2G) including charging infrastructure (Managed Charging V2.0). Technische Universitätsgesellschaft Hannover (2011)


Co-ordinating Protocol Interleaving for Coherent Conversations

Wilson Wong¹, Lin Padgham¹, John Thangarajah¹, and Wei Liu²

¹ School of Computer Science and Information Technology, RMIT University, Melbourne, Australia
{wilson.wong,lin.padgham,john.thangarajah}@rmit.edu.au
² School of Computer Science and Software Engineering, The University of Western Australia, Perth, Australia
wei.liu@uwa.edu.au

Abstract. This paper presents a mechanism for using protocol specifications to co-ordinate the independent dialogues of agents within a multi-agent implementation of a virtual character, to achieve a single coherent conversation. The virtual character interacts with a user via spoken dialogue, in a mixed initiative conversational mode. Activities are provided by different agents within the system which are independent of each other. In order to produce a single coherent conversation with the user, some co-ordination is required. We show how this co-ordination can be achieved by having each agent simply register the protocol specifications for the dialogues they can have with the user, along with associated message definitions. An interactor agent then uses these specifications completely transparent to the agents, to coordinate their dialogues with the user to produce a single, coherent user interaction. This allows modules providing functionality for the virtual character to be developed independently, considering only their own conversational activities. Basic co-ordination is achieved using simple stack management based on the protocol specifications, while a more advanced version of the interactor provides smoother conversation.

Keywords: Dialogue management; Multiagent system; Conversational agent; Interleaving protocols;

1 Introduction

Multi-agent systems rely on communications between agents to fulfill the tasks of the system. Specifying the patterns of interactions between particular subsets of agents in order to accomplish specific tasks is an important part of the design and specification of a multi-agent system. These protocol specifications are written in some form of (often graphical) language such as Agent Unified Modelling Language (AUML) [7] sequence diagrams. In this paper we show how a coordinating agent can use such specifications in order to assist in the management
of messages and the underlying dialogues, in the absence of clear information regarding which agent or dialogue a message is directed towards.

The particular class of application we are targeting is one where a human user is interacting with the multi-agent system via a virtual character and a speech interface. This system has a variety of abilities, controlled and provided by separate agents which are loosely coupled, allowing for the addition of new abilities to the virtual character [5], possibly developed by external providers. This scenario is motivated by a project to develop an intelligent Toy, where capabilities of the Toy, such as storytelling, math games or various interactive speech based activities can be developed and added to the Toy as plug-in modules.

In order for the Toy to communicate with the user as a single entity, there must be some co-ordination of the agents providing the capabilities of the Toy. However, we want these agents to be loosely coupled so that capabilities can be developed independently. One constraint imposed by the nature of the virtual character application is that the interactions that the various agents (offering different activities) have with the user must form a single coherent conversation. In this work we define a *conversation* as the sequence of interactions between the user and the virtual character during a single session. We define a *dialogue* as the interactions a particular agent has with the user around a specific *conversational activity*. The dialogues with different agents may be interleaved to form a conversation, but this should be done in a way that is transparent to the user.

We show how *protocols* or dialogue specifications, can be used as a mechanism together with a management agent, which we call the *interactor*, to orchestrate the interleaving of dialogues to achieve a co-ordinated and coherent conversation between the child and the Toy. This approach enables agent or module designers to develop their functionality for interacting with the user in isolation from other modules. By simply exposing the protocol specification and associated message types, a module can then be integrated into the Toy to contribute to conversations with the user.

In the following sections, we describe the interactor and the way in which it co-ordinates the dialogues during a conversation using protocol specifications provided by the participating agents. Importantly the participating agents require no knowledge of the co-ordination and do not need to consider this in their individual design. We first provide a brief introduction to protocol specifications, in particular AUML in Section 2. In the same section, we describe the multi-agent architecture of the Toy, illustrating the kind of interactions that would result if all agents acted autonomously without co-ordination in their interactions with the user. In Section 3, we describe the basic interactor agent that uses a stack to keep track of dialogues and performs only basic stack management tasks in order to achieve reasonable co-ordination and conversational coherence. In Section 4, we describe several advanced features that can be introduced to the interactor to further enhance the coherence of dialogue interleaving, while still being transparent to the participating agents. We motivate the description in each section with example conversations. We finish with a brief summary of the advantages of this approach and directions of future work in Section 5.
2 Background

An interaction protocol between a group of agents specifies all possible legitimate sequences of messages between the agents. One approach to protocol specification is that used within the work on Electronic Institutions, (e.g., [3]) where interactions are specified in terms of finite state machines. The other common approach to protocol specification is AUML sequence diagrams [1, 7], which are used in many of the popular agent development methodologies such as Prometheus, Tropos, O-MaSE and GAIA (see [2]). We briefly describe this notation in the context of protocols used within the Toy architecture.

2.1 AUML Protocol Specification

Protocols are widely used in design and specification of agents, as they are interface specifications, requiring no assumptions about the internal decision making or architecture of the participating agents, but do provide more structure than a simple API.

Agent UML (AUML) is an extension to the standard Unified Modelling Language (UML) and is intended to support the full software development cycle of agent systems. AUML sequence diagrams, also called protocol diagrams, illustrated in Figure 1, specify allowable sequences of messages between agents (or agent roles) with respect to a particular task, activity or conversation type. Specifications consist of vertical lines identifying agents or roles, horizontal directed arrows representing messages, labelled with the message type, and boxes representing a range of control constructs such as loop, choice (alt), optional (opt) and parallel (par). These control constructs can also have a guard which describes a relevant condition such as the termination condition for a loop, or the condition for a choice. Dashed lines separate options in choice or parallel boxes. Protocols can also include sub-

Fig. 1. The story selection protocol (SS) of the story teller agent. The number in brackets on messages are for easy referencing.
protocols using the \textit{ref} construct. Although protocols are usually shown graphically, there are also tools which use a textual representation to produce the figure (e.g. [11]). For a full specification of AUML protocols see [7].

Figure 1 shows an example protocol from the story teller agent within the Toy. The name of the protocol is in the top left corner. This protocol captures a loop of story suggestions (or alternatively notification that there is no story), which can be responded to with an \texttt{accept}, \texttt{reject} or \texttt{cancel} message from the user. In the optional box at the start of the protocol, we see an illustration of a guard, providing explanation of when this optional interaction will occur. We will not use guards in this work, but may include them to aid in understanding the intent of the protocol.

2.2 The Toy Conversational System

![Diagram showing the basic multi-agent setup to the Toy conversational system.]

\textbf{Fig. 2.} A basic multi-agent setup to the Toy conversational system.

Figure 2 shows the basic architecture of our virtual character, controlled by multiple agents. Each of these agents have some number of \textit{conversational activities} which they use to engage with the child. Some activities (e.g. weatherman) are part of the \textit{core} of the Toy along with the infrastructure support for speech recognition and speech synthesis, as shown in Figure 2. Most of the activities are provided by third party plug-in modules (e.g. meal reminder, story teller and math guru), conceptualised as agents, although the internal implementation does not have to be agent-based. The conversational activities offered by these agents are described in the form of protocols, as in Figure 3.

In the absence of any co-ordination provision, we might assume that each agent autonomously listens to the user, responding as appropriate when it hears a message to start or continue a particular conversation. In FIPA ACL\textsuperscript{3}, messages include a tag which identifies the dialogue a message belongs to. Although

\textsuperscript{3} \url{www.fipa.org/specs/fipa00061/SC00061G.html}
conversation identifiers are not often supported in agent platform infrastructure, it is relatively straightforward to ensure that computer agents do tag their messages in this way, if this is required by the application. Human users, however, do not reference their messages in such a manner, but rather expect the information to be garnered from the context. It is conceivable that each agent correctly recognises requests to start one of its conversational activities, and also recognises continuation messages. However, if two (or more) dialogues are active (perhaps due to the user changing their mind about what they wish to do), then there is potential for the ensuing conversation to become confused. Figure 4 gives an example of the kind of confused conversation that could result from agents individually having the conversational activities specified in the protocols in Figure 1 and 3, and responding according to these, without any co-ordination. The bracket at the end of each utterance relates the user input or system output to the associated protocol and message sequence. For example, MQ-1 refers to protocol MQ, which is in Figure 3(b), and the first message request-math-quiz.

### 2.3 Example of Poor Coherence

The example conversation starts with a user input on line 1, which is recognised by the math guru agent as a message of the type request-math-quiz, resulting in an initiation of a math quiz dialogue. Given that the topic on subtraction was already provided as part of the first message, the following two messages in the opt box are not exchanged. The dialogue progresses with the math guru sending the next message math-question (MQ-4) producing the system output on line
At this point, the meal reminder, triggered by its internal clock, also initiates a dialogue and outputs the `remind-meal` message. The user then changes their mind about what they wish to do and on line 3 indicates that they want a story about ducks, which is recognised by the story teller agent which starts a story selection dialogue and in line 4 outputs a `suggest-story` message (SS-5). When the user responds with “No. Not that.” on line 5, we get our first confusion caused by a single user input going to multiple dialogues as a valid response. The input on line 5 is recognised by two agents, the meal reminder who interprets it as a message of type `cancel` (MR-3) and the story teller who interprets it as a message of type `reject` (SS-7).

As a result, the system produces two distinct outputs from the two agents on line 6, which appear out of place when put together. The next user input on line 7 was not understood by any of the agents, causing the conversation to stall. At the same time, the story selection dialogue is pending user input after its last output on line 6. The meal reminding dialogue, after sending the last message (MR-4), is complete. The user, wanting to continue with the story activity, issued an input on line 8. This input is not a valid response to the story suggestion dialogue (which should be `accept`, `reject` or `cancel`). However, it is understood by the story teller as a message of type `request-story`, producing another instance of the same protocol on story selection, hence the system output on line 9. The same process of suggesting and either accepting or rejecting a story suggestion takes place between lines 10 and 12. The same problem of a user input being accepted by multiple dialogues occurs again on line 12 when the new story selection dialogue sees the “...” as a `reject` message (SS-7) and at the same time, the “…3 ducks” portion of the input is understood by the math guru as a
The interactor is essentially an intermediary agent between the user and the modules, whose primary tasks are to (1) initiate dialogues, (2) dispatch user inputs to the appropriate dialogue and relay outputs from the modules back to the user, and (3) deal with completed dialogues. This interactor agent exists within the core of the system as shown in Figure 6, and is a conduit for all messages. Figure 5 shows a revised version of our initial example conversation, produced with the support of a basic interactor agent.

![Fig. 5. A conversation mediated by the basic interactor](image-url)

![Fig. 6. An overview of the Toy system with the interactor agent as part of the core.](image-url)
3.1 Module Registration

In order for the interactor to be able to co-ordinate the potential multiple dialogues arising from different agents, we assume that each module (or agent) registers its protocols, and any custom message types. We assume that a message definition consists of a type name and a set of patterns defined using regular expressions, which specify the possible content of messages of this type.

The core has, along with speech processing ability, the ability to match spoken input to the provided patterns. It also has a set of generic message types, such as accept, reject and cancel which are exposed and can be used by modules within their protocols. During registration, protocols such as those shown in Figures 1 and 3 are provided in the textual form defined by Winikoff [11].

3.2 Interactor Agent

The basic functionality of the interactor agent is achieved by managing what we will refer to as the dialogue stack. The interactor is responsible for initialising a dialogue when required and adding it to this stack, removing dialogues when they complete, and re-ordering the stack as needed to ensure that the dialogue in focus is at the top of the stack. Figure 7 illustrates the transitions between the different dialogue stack configurations involved in the running example in Figure 5.

The three key tasks of the basic interactor agent for managing the dialogue stack are:

---

4 Message types are tagged with the agent ID to avoid any name clashes.

5 This functionality has been developed by the industry partner and is quite robust.
– **Dialogue initiation** is concerned with decisions on which protocols to be initialised and put on the stack based on inputs from the users or modules.
– **Dialogue focus management** deals with decisions on which dialogue should receive or retain focus, thereby having *temporary* control of the conversation.
– **Dialogue termination** deals with the removal of dialogues from the stack.

We explore these three key tasks, and demonstrate how they enable the basic coordinated conversation shown in Figure 5.

**Dialogue Initiation** A request to instantiate a protocol to create a dialogue can come from either the user, as in the case of *request-math-quiz* in Figure 5 line 1, or from a module as in the message *remind-meal* on line 2 in Figure 5.

**User-initiated dialogues** are started by user requests. Speech inputs from the user are first processed and then matched against the stored message patterns. The message type recognised and the actual input string are then provided to the interactor, which in the case of an initial message of a protocol, results in the initiation of a dialogue of that type that is then placed at the top of the stack. The message type and the input string are then passed on to the agent providing that protocol. Figure 7 shows examples of dialogues being initiated by the user and pushed onto the stack, in steps a and c.

**Module-initiated dialogues** occur due to the internal reasoning of agents, including both the third-party and the system utility modules. For example, the internal clock of the meal reminder module, or a storm warning retrieved online by the weatherman module could generate an initial message of the relevant protocol. Just as with user-initiated dialogues, this will be received by the interactor as a message type and specific string. The dialogue will be initiated, as in step b of Figure 7, and the message content passed to the speech output system for spoken delivery to the user.

**Dialogue Focus Management** When a new dialogue is started, it will be placed at the top of the stack. If the one in focus completes, then the dialogue immediately below it will be returned to focus. When an input message (i.e., a message type with content string, as recognised by the speech processing) is received, if it is not a request for a new dialogue, the normal case is that it is the next message in the dialogue currently in focus. The interactor checks the protocol specification to see whether in fact the message received is a valid message for the dialogue in focus, at this point in the dialogue. If it is, then the interactor passes the message (i.e., type and input string) to the relevant module, and updates its information as to where it is in the current dialogue.

---

6 Based on experience of the industry partner with actual interactions, we first check for requests for new dialogues. This is because initiation request message patterns are usually more specific than dialogue continuations, and if the continuation is checked first, it may well match, even though the input is actually an initiation request.
If however the message is not valid for the dialogue in focus, the interactor searches sequentially down the stack, looking for a dialogue where the message received would be valid. On locating the first dialogue where the message is valid for its current state, the interactor moves the identified dialogue to the top of the stack, giving it the focus, and sends it the message and input string. The step labelled $d$ in Figure 7 is an example of this. In this example, the math-answer message (MQ-5) does not match the input expected by the currently in focus story selection dialogue, but does match what is expected by the math quiz. The interactor therefore moves the math quiz dialogue to the top of the stack and forwards the message to the math guru.

In the event that a user input does not match the patterns of expected message types, the input will be ignored by the Toy system with this basic interactor, causing the conversation to stall. It may also be the case that the input message matches the start message of more than one protocol in which case the basic interactor will pick one of them at random to initialise. More sophisticated ways of handling these issues are discussed in Section 4.

In regard to module outputs, the bulk of the system utterances typically come from the dialogue in focus, as responses to the inputs passed through from the user. However, it is possible for modules to proactively send messages, as the meal reminder agent does in Figure 5 line 2. In such cases the interactor simply concatenates the different outputs and sends the single utterance to the user.

**Dialogue Termination** In the basic interactor, dialogues can be terminated by explicit user requests, module completion, or module request. For user requested cancellation, the dialogue in focus at the top of the stack is the one which is terminated. When a dialogue terminates, by completion or cancellation, the interactor removes the dialogue from the stack.

### 4 Enhanced Interactor

The basic interactor described in Section 3 ensures that only one dialogue receives a given user input, and that it is received by a dialogue which is expecting that input type. This is achieved through straightforward monitoring of protocols combined with management of a dialogue stack. The resulting improvement in coherence is obvious, but as can be seen in the conversation in Figure 5, there are still some places where there is room for improvement:

- **S-1:** The concatenation of 2 messages, in line 2, when the meal reminder module pro-actively sends a message, creates a somewhat awkward utterance.
- **S-2:** When a user input does not match a start message to a dialogue or the next message in existing ones, then the conversation stalls, as in line 7 of Figure 4.
- **S-3:** At line 8 in Figure 4, we see a user who wishes to return to an existing activity, which is no longer in focus. The input is detected as a new request for that activity instead and a new dialogue is started. As a result, the...
dialogue starts from the beginning whereas the user is expecting to pick up where they left off.

In addition to the above issues that can be seen in our example conversations, there are three additional issues we have identified:

S-4: When a dialogue finishes, if the conversation is to be returned to the next dialogue on the stack, the interactor can provide some information to indicate this to the user. Although it can be left to the user to direct the next part of the conversation, it can be advantageous for the interactor to take control. For example if at line 3 of Figure 8, the user had replied “No, I don’t want dinner”, this would end the meal reminder protocol, leaving the math quiz dialogue in focus. However, this would not necessarily be clear to the user.

S-5: When a user input matches the patterns of multiple start messages, one is chosen at random, but this may not be what the user intended.

S-6: When the stack is empty, either because it is the start of a session, or because all current dialogues have ended, the conversation may stall.

We now describe some additional components and strategies to address these issues, still in a fairly straightforward manner, without introducing any complex reasoning into the interactor. The two main components we introduce are:

– *utility protocols* that are used by the interactor to re-gain control of the conversation, or to clarify its direction; and

– a predefined set of *interactor messages* that the interactor can use, either to smooth the conversation, or as part of the utility protocols.

Figure 8 shows a conversation that is the result of an interactor with these additional capabilities, retaining many of the same user inputs as the two example conversations in Figures 4 and 5.

### 4.1 Interactor Messages

The enhanced interactor resolves many of the issues above by directly engaging the user, with no special requirements of the plug-in agents/modules. The interactor communicates with the user for this purpose using a range of custom messages constructed from some hand crafted text fragments. To ensure variability in the interactor’s outputs to the user, there can be multiple versions of a fragment that are semantically equivalent. These messages also contain two types of variables: (i) a `protocol-name`, and (ii) an `utterance`. The protocol names are available as part of module registration. The utterance is either the current output utterance from a module or the last utterance in some dialogue. The latter requires the interactor to record the last module utterance for each dialogue on the stack.

There are six types of interactor messages which support one or more of the utility protocols as follows:
USR: i learned subtraction today. test me. (MQ-1)

SYM: mama duck has 4 baby ducks. 2 went out, how many left? (MQ-4, IM, MR-1)

USR: i want a story instead about ducks. (SS-1)

SYM: i know 2 stories about ducks. want to listen to the ugly duckling? (SS-5)

USR: no. not that. (SS-7)

SYM: how about a story about the 3 proud ducks? (SS-5)

USR: ah. i remember. the answer is 2. (MQ-5)

SYM: that’s correct! (MQ-6)

USR: have you seen my barbie? (RCP-1)

SYM: i thought we were doing math quiz. two ducks... (RCF-2,MQ-5)

USR: yes. i think the answer is 3 (RCF-2,MQ-5)

SYM: i’m sorry. that’s not correct. how about this question... (MQ-7)

USR: that’s hard. i want my story back. (MQ-4)

SYM: let’s go back to story selection. how about a story about the 3 proud ducks. (RFF-1)

USR: ok. (SS-6)

SYM: once upon a time... (SS-ST)

...

SYM: that’s the end. (SS-ST)

USR: do you want to continue with the math quiz? (RFF-2)

USR: no. (RFF-5)

SYM: let’s go back to meal reminding? mum says it’s dinner time. (RFF-1)

USR: no. i don’t want dinner. (MR-3)

SYM: i’ll stop reminding you. do you want to do singing? (MR-4)

USR: no. (PP-3)

SYM: do you want to do drawing? (PP-1)

...

Fig. 8. A conversation mediated by the interactor with additional functionalities for comparison with Figure 5. IM refers to the one-shot use of the interrupt-message.

- **Interrupt message** is used when one module interrupts a dialogue by proactively producing an output utterance. An interrupt-message combines and decorates the two output utterances to form a single coherent system utterance as such:
  “$utterance1. By the way, $protocol-name wants to say $utterance2.” This addresses issue [S-1] above. Line 2 of Figure 8 shows an example use of this message type.

- **Recovery message** is used to respond to a user input that does not match any expected messages of the dialogues on the stack, or any start messages. The intent is to refocus the user back to the dialogue that is currently in focus, as in line 10 of Figure 8, using statements such as:
  “I thought we were doing $protocol-name? $last-utterance.”

- **Re-focus seamless message** is used as a single message to smooth the transition back to an inactive dialogue, possibly at user requests:
  “Let’s go back to $protocol-name. $last-utterance.”
– **Re-focus confirm message** is used to ask the user whether they wish to return to an inactive dialogue:

> “Do you want to continue with $protocol-name?”

This needs to be used as part of a protocol which handles the response. It is more appropriate than the **refocus-seamless-message** above if the dialogue has been inactive for a longer time.

– **Clarification message** is used to clarify which dialogue to initiate when a particular input matches multiple start messages. For simplicity a single option is chosen so that a yes-no response is easily recognised.

> “Do you want to do $protocol-name?”

– **Proposition message** is used to pro-actively suggest a conversational activity:

> “Do you want to do $protocol-name?”

This can be useful when there is nothing on the stack.

### 4.2 Utility Protocols

Utility protocols define brief dialogues the interactor can have with the user, making use of the newly defined interactor message types and user responses (which will be matched to core message types or protocol defined message types, as usual). These protocols are managed in exactly the same way as other protocols in the system, except that the interactor makes the decision to initialise the corresponding dialogue, based on the situation, rather than on a direct input from the user or module. The four utility protocols we define are as follows:

**Recovery protocol** (RCP), shown in Figure 9(a), is used by the interactor to react to an input that does not match the expected input of dialogues on the stack or any patterns to start a new dialogue, which is issue [S-2] described above. The **valid-message-type** refers to the message type expected by one of the dialogues on the stack, or a start message for a new dialogue, while **invalid-message-type** refers to a message that is not a **valid-message-type**. Lines 9-10 of Figure 8 show the use of this protocol. The interactor sends the first message, which is **recovery-message**, to describe to the user what the system thinks they are talking about and repeats the last utterance of that dialogue. The hope of sending this message is that the user will respond with an input that a dialogue on the stack can recognise. If this does not happen (and the response is also not a start message for a new dialogue), then the interactor will attempt to regain control by proposing a new activity (using the proposition protocol).

**Refocus protocol** (RFP), shown in Figure 9(b), is used by the interactor when returning to a dialogue that has been out of focus. The **last-utterance-message** refers to a message containing the last utterance from the dialogue to be returned focus. If the user has indicated a desire to return to a dialogue, or if the interactor decides to return without confirmation (perhaps because the inactivity has been short-lived), then the first alternative of a single **refocus-seamless-message** is used, which includes the last utterance. Lines 14 and 27 of Figure 8 illustrate
the use of this protocol to address issues [S-3] and [S-4], respectively. Alternatively the interactor first asks if the user wants to return to this dialogue, using the refocus-confirm-message. If the answer is accept, the last utterance is provided to restart the dialogue. If it is a reject, the dialogue is removed from the stack and the interactor moves on to attempt to re-focus the next dialogue on the stack. Lines 25-26 of Figure 8 show examples of this second option.

Clarification protocol (CP) in Figure 9(c) is used for clarifying user inputs which can be used to initiate multiple dialogues. This protocol simply iterates through the possible options asking if this is what the user wanted using a clarification-message. This addresses issue [S-5].

Proposition protocol (PP) shown in Figure 9(d) is used by the interactor when
it needs to be pro-active in ensuring the continuity of a conversation, either because the previous dialogue has ended or it is a new session. A more satisfactory approach would be to have a core agent whose task is to initiate activities, based on reasoning about the child’s interests, history, and so on. In that case the interactor would simply request this agent to initiate the desired dialogue. However, we provide here a simple protocol, using the proposition-message that could be used by the interactor as a simplified way of proposing a new activity. This addresses issue [S-6] and an example can be seen in lines 29-31 of Figure 8.

5 Conclusion

In this work, we present a new approach to co-ordinating dialogue between a human user and conversational modules within a multi-agent implementation of a virtual character for coherence. This approach involves an intermediary agent called the interactor, and the protocol specifications of the conversational activities, e.g., story telling, math quiz, offered by the participating modules.

We proposed two levels of co-ordination for the interactor, namely, basic and enhanced, each providing different levels of sophistication to produce single, coherent conversations, completely transparent to the agents in the system and the human user.

We showed through an example that reasonable coherence could be achieved by the basic interactor that performs only stack management tasks, without any reasoning overhead. This design is loosely related to that of Turunen & Hakulinen [9], where only a single dialogue agent can be active at any one time. In contrast to that approach, we allow for switching between multiple dialogues that may be at different stages of progress.

There were however several issues with the basic interactor that may still compromise the coherence of a conversation. To address some of them, the enhanced interactor was introduced with two main components; a set of utility protocols and custom crafted fragments. These, together with limited reasoning about dialogue duration and history, allows the interactor to perform more advanced tasks such as smoothing transitions between different dialogues and proactively engaging the user in resolving ambiguity.

The key advantage of our approach is that the only requirement for a particular module to contribute conversational activities to the system is to register the interaction protocols and any custom message types. This minimal requirement is also what distinguishes our approach from existing ones that use mediators for dialogue management [9, 4–6, 8, 10]. Our approach allows for the modules in the system to be loosely coupled, allowing external developers to build and integrate conversational modules independent of each other.

As future work, we look at how user preferences may be captured and used in managing conversations involving different activity modules. Currently, the onus is on the individual modules to keep track and make use of personalisation data. One possibility is to introduce into the interactor the ability to resolve issues based on user preferences. Another area on future interest, is integrating a
‘chatter’ agent (such as in [12]) to the core of the system providing the interactor and other modules the ability to obtain short text snippets to embellish outputs or to engage the user in brief chat sessions.

Acknowledgment

This work is partially supported by the Australian Research Council and Real Thing Entertainment Pty. Ltd. under Linkage grant number LP110100050.

References

Engineering Conflict-Free Multiagent Systems

Akın Günay and Pınar Yolum

Computer Engineering Department, Bogazici University, Istanbul, Turkey
{akin.gunay,pinar.yolum}@boun.edu.tr

Abstract. Organizations have served an important metaphor for designing and engineering multiagent systems. Usually, organizations are specified over abstract roles, which specify what the enacting agents should or should not do. In order to ensure that organizations work properly, we need to check that the roles have been designed correctly in the first place. Accordingly, we analyze various relations between roles to check if they include conflicting terms, which can prevent the enacting agents to conform with the obligations and prohibitions. We identify various types of conflict that may occur in a role specification and between different roles in an organization. In order to formalize the discussion we represent role specifications as sets of commitments. Then using the conflict relations between commitments, we define basic principles that should be followed in order to develop role-based conflict-free multiagent organizations.

1 Introduction

Organizations have been an important metaphor in understanding, designing, and engineering multiagent systems [1, 6]. Organizations allow division of tasks, assignment of these tasks to different individuals and provide a mechanism for disseminating task results between individuals. Many times, organizations are specified abstractly over roles, rather than individual agents. This allows a generic specification so that at different times different agents can be matched to various roles and realize the organization as a multiagent system.

Organizations exist to carry out specific tasks or reach specific goals. A correct specification of an organization should guarantee that the organization works in harmony and the overall goals of the organization are achieved. In order to achieve this objective the specification of the organization should be conflict-free. Since roles are the main building blocks of organizations, the first step of designing a conflict-free organization is to ensure that the roles in the organization are themselves conflict-free and the agents that attempt to carry out the roles are not prevented from satisfying the requirements of the roles out of their will. For example a program committee member role in a conference organization should not concurrently require and prohibit the enacting agent to review a paper. This would be conflicting and would prevent the agent to fulfill the requirements of the role event if it intends to do so, which in turn causes also the organization to fail to achieve its objective.

* This work is partially supported by Boğaziçi University Research Fund under grant BAP5694, and the Turkish State Planning Organization (DPT) under the TAM Project, 2007K120610.
Besides the conflicts within the role specification, another issue that may prevent an organization to work in harmony arises when agents are allowed to enact multiple roles simultaneously. In this case it is not enough only to ensure that the roles themselves are conflict-free, but also the specifications of the roles that may be enacted by the same agent should be conflict-free with each other. If an agent enacts to two roles at the same time, it should not be required to bring about a condition as part of the first role and required not to bring about the same condition as part of the second role. For example in a conference organization agents usually enact both to program committee member and author roles simultaneously. Program committee member role obligates the enacting agents to review assigned papers. On the other hand author role prohibits the enacting agents to see papers of other authors. When the mentioned obligation and prohibition of these two roles are considered, it makes the impression that these two roles should not be enacted by the same agent. However, these two roles actually can be enabled to the same agent by allowing the agent to access only to certain papers that are assigned to it. But, if this issue is not precisely specified, then the roles may conflict with each other by obligating the agent to review papers while prohibiting it from accessing to those papers.

Even if the roles in an organization are specified in a conflict-free manner while designing the organization, problems may still arise at run-time due to certain affairs that cannot be foreseen at design time. A major problem, which prevents the organization to function properly, occurs at run-time when the enacting agents suffer from lack of resources that are required to fulfill the obligations of their roles. For example, if five papers would be assigned to an agent enacting the program committee member role and the agent would have time only to review three papers, then the agent would fail to fulfill its obligation. In some organizations role specifications may include expected resource requirements of the role, which can be used by agents to check their compliance before enacting a role. However, making such predictions is usually challenging due to open nature of multiagent systems. For example, the specification of program committee member role may state that three papers are expected to be assigned to enacting agents for review. However, if the total number of submission would exceed the prediction that had been made while designing the system, more papers might be assigned to program committee members. Besides, in many situations it is not even possible to make such predictions about resources (e.g., scheduling of events).

Various works exist where multiagent systems are engineered through their specification as organizations [4, 24]. Once an organization has been specified, the general focus has been on assigning agents to particular roles. Most of these approaches rightly focused on whether the agent has the necessary capabilities to bring about the requirements of the roles. Aiming to create more robust and conflict-free multiagent organizations, here, we consider various other aspects that might cause to conflicts in designing and infeasibility while executing these organizations. To do so, we represent obligations and prohibitions of roles of an organization through commitments. We identify situations where commitments would conflict thereby leading to a role conflict. Similarly, we use the feasibility concept of commitments to capture infeasible situations at run-time and we discuss several methods to deal with such situations.
The rest of this paper is organized as follows. Section 2 gives the necessary background on commitments and describes how roles are denoted through commitments. Sections 3 and 4 develop what it means for roles to conflict and how they can be resolved. Finally, Section 5 discusses our work in relation to recent literature.

2 Technical Background

In the rest of the paper we use propositional symbols to represent domain dependent concepts. We assume that these concepts are defined in a domain ontology that is available to all agents. We use the logical connectives $\land$ and $\rightarrow$ in their common semantics and $\neg$ as logical negation. We use the variables $x$, $y$ for agents, $p$, $q$, $u$, $w$ for propositions, $c_i$ for commitments and $r_i$ for roles. Constant symbols start with capital letters. In general we use a conference organization as a running example, as it is widely used in the previous work about organizations. Besides, to make our discussion more refined in some examples we also use some other related domains when these domains are more intuitive.

2.1 Commitments

$\text{C}(x, y, q, p)$ denotes a commitment from the debtor agent $x$ to the creditor agent $y$ to bring about the consequent $p$, if the antecedent $q$ holds [22]. The antecedent of a commitment can be a conjunction of propositions, however we assume that the consequent of the commitment is a single proposition. Let us consider the following example from the conference organization domain. In this domain a member of conference’s program committee ($\text{PCMember}$) is committed to review a paper ($\text{PaperReviewed}$), if the program chair ($\text{Chair}$) assigns the paper to the program committee ($\text{PaperAssigned}$). We denote this using the following commitment:

$\text{C}(\text{PCMember}, \text{Chair}, \text{PaperAssigned}, \text{PaperReviewed})$

A commitment is a dynamic entity with a lifecycle [7, 17, 22]. A commitment’s state evolves as the result of certain actions that are performed by the participating agents. In order to focus on our contributions, in this paper we use a simplified lifecycle of commitments as follows: A new commitment is created by its debtor using the $\text{Create}(x, y, q, p)$ action, which creates the conditional commitment $c = \text{C}(x, y, q, p)$. When the antecedent of the commitment starts to hold, the creditor performs $\text{Detach}(c, q)$ action that makes the commitment active. To represent an active commitment we replace the antecedent of the commitment to $\top$ symbol (i.e., $\text{C}(x, y, \top, p)$). When the consequent starts to hold, the debtor performs $\text{Discharge}(c, p)$ action that makes the commitment fulfilled. On the other hand, if the debtor cancels its commitment using $\text{Cancel}(c)$ operation, then the commitment is canceled. In addition to committing to bring about a condition, agents may also commit to maintain a condition [2, 13, 23]. For instance, the web admin of a conference should keep the Web site of the conference running. We represent such commitments by drawing a line on top of the corresponding condition.
Such commitments cease to exist only when the creditor of the commitment $c$ releases the debtor by performing the $\text{Release}(c)$ action.

2.2 Role Specifications

A role describes what an agent should and should not be doing. For instance, an agent in the program committee member role should review assigned papers by the program chair. This can be seen as the agent’s obligation. On the other hand, the same agent should not be reviewing a colleague’s paper. This can be seen as a prohibition. We represent each such clause (i.e., obligation or prohibition) as a separate commitment. Considering the program committee member role the obligation to review assigned papers can be represented using the following commitment:

$$C(\text{PCMember}, \text{Chair}, \text{PaperAssigned}, \text{PaperReviewed})$$

Similarly, the prohibition to not to review colleagues papers can be represented using the following commitment:

$$C(\text{PCMember}, \text{Chair}, \text{AuthorIsColleague}, \neg \text{PaperReviewed})$$

A role exists as part of an organization committee role exists because a conference is being organized. Depending on the progress of this organization, each role can be active or passive. The state of the roles are triggered by some conditions. For instance, the program committee member role becomes active when the paper submission is closed. That is, the requirements of the roles will apply on after this time. Similarly, the role becomes passive after the authors are notified about the acceptance of papers. From this point on, the role is not binding.

Definition 1 (Role). A role is a tuple $\langle \lambda, L, C \rangle$, where $\lambda$ is the unique label of the role, $L$ is formulae to define the conditions that activate and passivate the role, and $C$ is a set of commitments that the enacting agent should create.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>PCMember</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$ Activate:</td>
<td>${\text{SubmissionClosed}}$, Passivate: ${\text{AuthorsNotified}}$</td>
</tr>
<tr>
<td>$C$</td>
<td>$C(\text{PCMember}, \text{Chair}, \text{PaperAssigned}, \text{PaperReviewed})$</td>
</tr>
</tbody>
</table>

| $C$ | $C(\text{PCMember}, \text{Chair}, \text{AuthorIsColleague}, \neg \text{PaperReviewed})$ |

Table 1. Specification of the program committee member role.

We present a simplified role specification for a program committee member in Table 1. The unique label of the role is PCMember. The role is activated when the paper submission is closed and passivated when the authors are notified about the results. The program committee member is obligated to review assigned papers by the program chair.
The program committee member is prohibited to review papers if an author is also a colleague. Note that our role specification does not require any particular organization structure. It can be used as part of any organizational schema that has roles as part of it.

3 Role Conflicts

While acting in a multiagent system, an agent usually participates in multiple commitments at the same time. In such situations, the commitments of an agent may conflict. For instance, if a program committee member agent would be (somehow) obligated and prohibited to review a paper simultaneously because of its two different commitments, these commitments would cause a conflict and as result of this conflict the agent would be unable to fulfill both commitments. In other words the agent would inevitably violate one of its commitments because of the conflict between the commitments and nothing that the agent can do can fix this. This is an undesired situation and should be avoided whenever possible. In this section we first give a formal definition of such commitment conflicts. Then in the rest of the section we discuss such conflicts in the context of role specifications.

As the above example demonstrates, a conflict occurs between two commitments of an agent when the consequences of the commitments are inconsistent with each other and hence cannot be brought about at the same time. In this context we provide a simplified version of a commitment conflict between two commitments as follows [9].

Definition 2 (Commitment Conflict). Given two (active or conditional) commitments $c_i = C(x, y, w, u)$ and $c_j = C(x, y', w', u')$, $c_i$ and $c_j$ conflict with each other, denoted by $c_i \otimes c_j$, if:

- it is not the case that $w \rightarrow q$ and $w' \rightarrow \neg q$, and
- it is the case that $u \rightarrow p$ and $u' \rightarrow \neg p$.

In order for two commitments to conflict, their consequences should be inconsistent. That is, while one commitment is telling the agent to do one thing, the second commitment should say the opposite. We capture this by requiring that while one consequent implies a proposition, the other one implies the negation of the same proposition. The second condition in the previous definition captures this. If both commitments are active commitments (i.e., their antecedents already hold), then this is the only condition that needs to be checked. However, if either or both of the commitments are in the conditional state, then first a check is necessary to see if both commitments can actually end up in active state. For this to happen, the antecedents of both commitments should not be inconsistent; that is they should be able to hold at the same time. This will enable both commitments to become active at the same time. The first condition in the previous definition ensures this.

3.1 Conflicts in a Role Specifications

Development of a role-based organizational model requires accurate analysis of obligations and prohibitions that are associated with each role. However, such analysis is
a challenging task and may create conflicts if not done properly. If a role is not designed carefully and the specification includes conflicting obligations and prohibitions (i.e., conflicting commitments), then the enacting agent ends up in a situation where it inevitably violates its commitments, which in turn affects proper working of the organization. For instance, consider again the program committee member role and the associated commitments to review assigned papers and not to review colleagues’ papers, which are represented as follows:

\[ c_1 = \text{C(} \text{PCMember, Chair, PaperAssigned, PaperReviewed) } \]

\[ c_2 = \text{C(} \text{PCMember, Chair, AuthorIsColleague, } \neg \text{PaperReviewed) } \]

These two commitments are in conflict \((c_1 \otimes c_2)\), since the consequences are inconsistent, while the antecedents are not. Once a paper is assigned to a program committee member (i.e., PaperAssigned holds), then \(c_1\) is detached. Hence, the program committee member is obligated to review the paper (i.e., committed to bring about PaperReviewed). On the other hand, if the author of the assigned paper is a colleague, then \(c_2\) is also detached (i.e., AuthorIsColleague holds). Hence, the program committee member is prohibited to review the paper (i.e., committed not to bring about PaperReviewed). When both commitments become active, then the conflict between the commitments prevents the program committee member to fulfill \(c_1\) and \(c_2\) simultaneously.

This conflict occurs due to the imprecise description of the commitments’ antecedents. The above conflict can easily be avoided by making the antecedents more specific.

\[ c'_1 = \text{C(} \text{PCMember, Chair, PaperAssigned} \land \neg \text{AuthorIsColleague, PaperReviewed) } \]

\[ c'_2 = \text{C(} \text{PCMember, Chair, PaperAssigned} \land \text{AuthorIsColleague, } \neg \text{PaperReviewed) } \]

The modified antecedents are inconsistent (i.e., cannot hold at the same time). Hence, the commitments do not conflict anymore. More specifically, if the author of the assigned paper is not a colleague, then the antecedent of \(c'_2\) fails to hold. Hence, the only active commitment can be \(c'_1\). On the other hand, if the author of the paper is a colleague, then the antecedent of \(c'_1\) fails to hold and the only active commitment can be \(c'_2\).

The above conflict is a result of poor analysis of the situation. However it can be resolved with certain modifications of the commitments. On the other hand, some fallacies in the analysis may lead to fatal cases in which conflicts cannot be avoided. Such situations usually occur when a role is overloaded with obligations and prohibitions that actually should not be a part of the role. For instance, in a blind-review process the program committee member is prohibited to see author information of the papers. Assume that the program committee member role is analyzed inaccurately and the task to assign reviewers to submitted papers, which is normally a task of the program chair, is associated with the program committee member role. In this case the following commitments are part of the role specification of the program committee member:

\[ c_1 = \text{C(} \text{PCMember, Author, PaperSubmitted, ReviewerAssigned) } \]

\[ c_2 = \text{C(} \text{PCMember, Author, } \top, \neg \text{AuthorInfoAccessed) } \]
Note that in order to assign a paper, the program committee member should see the author information of the paper (i.e., ReviewerAssigned → AuthorInfoAccessed). Hence, the propositions PaperAssigned and ¬AuthorInfoAccessed are inconsistent. Accordingly, $c_1$ and $c_2$ do conflict with each other. Different than the previous example, here it is not possible to avoid this conflict by modifying the commitments, which indicates that the existence of one of these commitments in the context of the program committee member role is conceptually wrong. That is $c_2$ should not be a part of this role specification in the first place. The above discussion leads us to the following result.

**Definition 3 (Incoherent Role Specification).** A role specification $r$ is incoherent, denoted by $I(r)$, if there is a conflict between two commitments that are part of the role. Formally:

$$\exists c_i, c_j : c_i \in r.C \land c_j \in r.C \land c_i \otimes c_j \Rightarrow I(r)$$

**Conjecture 1** If a role specification is incoherent, then the agent that enacts the role cannot fulfill all commitments.

Conjecture 1 points out to a conceptual problem in the design of the incoherent role, which should be fixed either by modifying the commitments or reconsidering the involvement of some commitments as part of the role specification.

### 3.2 Conflicts between Roles

In many situations agents in an organization enact more than one role at the same time. For instance, in the conference organization an agent that enact to the program committee member role may also enact for the paper author role at the same time. In such situations, even if the role specifications enacted by the agent are individually coherent, conflicts may still arise between the agent’s commitments that belong to different roles. Assume that an agent enacts both the program chair and program committee member roles at the same time. The program chair role involves the obligation to assign a reviewer to the submitted papers as represented by the following commitment:

$$c_1 = C(Chair, Author, PaperSubmitted, ReviewerAssigned)$$

On the other hand the program committee member role prohibits the enacted agent to see the author information of papers as the following commitment states:

$$c_2 = C(PCMember, Author, \top, \neg AuthorInfoAccessed)$$

As we have mentioned in the previous section, these two commitments conflict since assignment of a paper to a reviewer requires the agent to access the author information of the paper which is inconsistent with the prohibition of the program committee member role to access author information of papers. Assuming that the program chair and program committee member roles are coherent in themselves, such a conflict indicates that these two roles should not be enacted by the same agent at the same time. We call such roles as mutually exclusive roles.

However, in many real life situations conflicts between the commitments of different roles may not always require the roles to be mutually exclusive. For instance, a
program committee member is allowed to see the contents of the papers she has been assigned. On the other hand, an author should not be seeing the contents of others’ papers. However, if an agent enacts to both roles (assuming the agent is qualified for both), the agent can actually access the content of other authors’ papers without violating its role as an author. In this case the commitment of to the program committee member overrides the commitment of to the author role. Hence, the prohibition of the author role to not view others’ papers is not valid anymore. When such a case is a part of the modeled organization, then the override relation of the commitments should be explicitly represented.

Definition 4 (Override Relation of Commitments). Given two commitments $c_i$ and $c_j$, $c_i \gg c_j$ denotes that $c_i$ overrides $c_j$.

Definition 5 (Mutually Exclusive Roles). Two roles $r_i$ and $r_j$ are mutually exclusive, denoted by $r_i \triangleleft r_j$, if there is a conflict between at least one commitment of each role and there is no override relation between the conflicting commitments. Formally:

$$\exists c_i, c_j : c_i \in r_i \land c_j \in r_j \land c_i \otimes c_j \land c_i \gg c_j \land c_j \gg c_i \Rightarrow r_i \triangleleft r_j$$

Conjecture 2 If two roles are mutually exclusive, then these roles cannot be enacted by the same agent simultaneously.

Again, understanding whether two roles are mutually exclusive can be detected during design time. Since, an agent that is enacting mutually exclusive roles are doomed to violate some of its commitments, it is wise to design the organization to avoid mutually exclusive roles.

4 Run-time Feasibility

The reason of the conflicts we describe in the previous section is the inconsistency of the commitments (i.e., obligation and prohibition of an agent simultaneously to bring about the same condition). Such conflicts should be avoided at design time in order to come up with a conflict-free multiagent system.

Even though a multiagent system is designed free of conflicts by making the role specifications coherent and by restricting enactment of mutually exclusive roles by the same agent, agents may still face up with challenges at run-time when they enact the roles. One important reason for this is the lack of resources needed to satisfy the commitments that the agents are involved in. Such situations arise when the commitments of an agent are infeasible [10].

In general, a role specification covers the capabilities of an agent that are required to enact the role. However, the actual resources that are required while performing these capabilities are usually revealed only at run-time\(^\dagger\). For instance, remember the commitment of the program committee member role to review the assigned papers. By the role specification it is clear that the agent that enacts this role should have the capability to

\(^\dagger\) We assume that the resource types and their amounts that are required to perform certain capabilities are represented in a domain ontology.
review an assigned paper. However, the actual number of papers that may be assigned
is not pointed out in the role specification. If the agent would be capable to review only
a certain number of papers and the number of assigned papers would exceed this limit
at run-time, then the commitments of the agent would be infeasible and the agent would
have no way to fulfill its commitments.

Note that in some situations the amount of resources required while enacting the
role can be defined at design-time. For instance, the maximum number of papers that
would be assigned to a program committee member agent might be defined in the role
specification. However, due to the open nature of multiagent systems making such as-
sumptions is usually not possible. For instance, the actual number of submissions at
run-time might exceed the number that has been predicted at design-time. In such a
situation more papers than it is defined in the role specification might be assigned to
program committee member agents.

The issue is even more challenging if temporal constraints are introduced to the de-
signed multiagent system. For instance, consider an agent that plays a professor role.
The agent is obligated to conduct research and submit papers to conferences to pub-
lish results. The agent is also obligated to attend a conference to make a presentation,
if a submitted paper is accepted. These obligations are represented by the following
commitments in the professor role specification.

\[ c_1 = c(\text{Professor, Department, } \top, \text{PaperSubmitted}) \]
\[ c_2 = c(\text{Professor, Department, PaperAccepted, ConferenceAttended}) \]

Another obligation of the professor role is giving lectures, if a lecture is assigned.

\[ c_3 = c(\text{Professor, Department, LectureAssigned, LectureGiven}) \]

Assuming that the agent cannot attend to a conference and give a lecture simultane-
ously (i.e., ConferenceAttended and LectureGiven cannot hold at the same time) one of
\( c_2 \) or \( c_3 \) would be violated by the agent, if these two events are scheduled to the same
time. However, it is not possible to determine exact times of these events while design-
ing the professor role. On the other hand, it is also not rational to remove one of \( c_2 \) or
\( c_3 \) from the role specification, only because they may introduce a potential infeasibility
at run-time. Nevertheless, in order to build a solid multiagent system, both the agents
that aim to enact roles and the design of the multiagent system should be fitted out with
mechanisms in order to deal with infeasibility.

In the rest of this section we first give a definition of feasibility adopting the defi-
nition we have developed in our previous work [10]. Then we define rules-of-thumb in
the context of roles and commitments to deal with infeasibility, once it is detected.

**Definition 6 (Running Multiagent System).** A running multiagent system \( m \) is a two-
tuple \( \langle A, C \rangle \), in which \( A \) is a set of agents and \( C \) is a set of commitments created by the
agents in \( A \).

Note that we do not mention the roles in the definition of a running multiagent
system. However, we assume that the agents in the system create commitments in ac-
cordance with the role specifications they enact following to the role based design of
the multiagent systems.
Definition 7 (Snapshot). $s_t$ is a snapshot of the commitments in $m.C$ and their states in a running multiagent system $m$ at moment $t$.

Hence, from a snapshot one can induce what commitments have been created, which one have been fulfilled, and so on.

Definition 8 (Commitment Feasibility). *In a running multiagent system* $m$, a given set of active or conditional commitments $C \subseteq m.C$ in snapshot $s_t$ is feasible, if there is a snapshot $s_{t'}$ in which every commitment in $C$ is fulfilled and $t < t'$.

The feasibility definition states that a commitment set is feasible at a given moment, if the multiagent system may progress in such a way that all commitments in the initial set are fulfilled. A multiagent system progresses based on the rules of commitment lifecycle. For example, an active commitment can either be fulfilled or violated, but a violated commitment can never become fulfilled. By generating possible future snapshots, one can infer it is possible for all the commitments to be successfully fulfilled. Hence, feasibility does not guarantee fulfillment of commitments, but only states that this is possible. On the other hand, once infeasibility of commitment is detected, then it is certain that the commitments cannot be fulfilled as they are.

Achieving Feasibility by Delegation: In many organizational models (e.g., hierarchies) agents who enact certain roles may have the power to delegate some of their responsibilities to other agents [12]. For instance, a professor may delegate lecturing duty to a teaching assistant (TA). In terms of commitments, delegation can be performed using the Delegate($c$, $a$) operation, which states that the commitment $c$ is delegated to the agent $a$. This operation is equal to releasing $c$ and creating a new commitment with identical conditions, but new debtor $a$ [17]. Although delegation provides an intuitive mechanism to resolve infeasibility, it should be used carefully. In this context, two major issues should be considered. The first issue is the problem to precisely define the delegation power over roles. This can be achieved by introducing new commitments to define delegation power. Although the same functionality can be achieved by expanding the notion of role specifications, commitment based approach is advantageous, since it eliminates introduction of new structures to the role specifications. For instance, in the specification of a teaching assistant role there might be a commitment, which states if a commitment is delegated (CommitmentDelegated) by the Professor (actually the agent who enacts this role), then the teaching assistant is committed to accept the delegation, i.e, create the delegated commitment (CreateCommitment).

$C(TA, Professor, CommitmentDelegated, CreateCommitment)$

Note that in our conference organization examples in the previous sections, assignment of a submitted paper from the program chair to a program committee member for review is also kind of a delegation. However, in that case the details of the delegation is explicit and well defined in the role specification as part of the conceptual design of the organization. The delegation mechanism that we discuss here is a loose one. In the above example, the professor is free to delegate any commitment to the teaching assistant, which takes us to the second important issue about the delegation. That is whether
the deputy agent (e.g., the teaching assistant) is capable to fulfill the delegated commitment. Two things should be considered in this context. First, it is necessary to determine whether the capabilities of the deputy agent is compatible with the requirements of the commitment in order to fulfill it. For instance, if the teaching assistant is not capable of giving lectures, then the delegation does not resolve the infeasibility. Second, it is necessary to check that the result of the delegation does not cause to another infeasibility, due to lack of resources of the deputy agent. For instance, if the teaching assistant is scheduled for another event at the lecture time, then the infeasibility persists.

**Conjecture 3** If the commitments of an agent are infeasible and the agent has delegation power to resolve the infeasibility, then the agent should delegate one or more commitments, only if the deputy or deputies are capable of and have enough resources to fulfill the delegated commitment(s).

As generally accepted in multiagent systems, the capabilities and state (e.g., existing resources to the agent) of an agent are private to that agent. Hence, in general the delegating agent may not know whether the deputy is capable of or has enough resources to fulfill the delegated commitment. Therefore, in order not to create a new infeasibility as a result of the delegation, an interaction should be carried out between the delegating agent and the deputy before the delegation.

**Compensation of Violation:** Delegation may not be used always to deal with infeasibility (e.g., the agent may not have power for delegation, etc.). In such situations, compensation may be used as an alternative method to expel the noxious effects of violation [19, 21]. In compensation, when a commitment is violated, another commitment (usually including a sanction) is created by the debtor to compensate the violation of the earlier commitment. Although compensation does not prevent violation, it still provides a mechanism to bring the system back into a healthy state. For instance, if the conference and the lecture of the professor would be scheduled for the same time and accordingly the professor would violate its commitment, the professor might commit to give another lecture at a later time to compensate the missing lecture. However, as in the case of delegation, it is necessary to take into account another infeasibility that may occur due to creation of the commitment for compensation (e.g., new schedule of the lecture for compensation may cause to another infeasibility).

Compensation can be performed using the commitment operation \( \text{Compensate}(c_i, c_j) \), which states that if the commitment \( c_i \) is violated then the commitment \( c_j \) is created as a compensation. As in the case of delegation, compensation can also be represented as a commitment in the role specification, without introducing a new notion. For instance, the compensation of a missing lecture can be represented using the following commitment:

\[
C(\text{Professor, Department, LectureMissed, ScheduleLectureToCompensate})
\]

In this example, \( \text{LectureMissed} \) holds when the professor’s commitment to give the lecture is violated and \( \text{ScheduleLectureToCompensate} \) holds only when a new commitment is created by the professor to give another lecture for compensation.
Note that in this example the commitment that will be created for compensation is well defined at design time, which is scheduling of another lecture. In some situations it may not be possible to determine a compensating commitment for each possible violation while designing the role specification. In such situations it may be a good practice to integrate a generic commitment, which states that when a commitment is violated then a compensating commitment should be created, into role specifications to enforce the agents to compensate their violated commitments. However, in such a situation context of the compensation should be agreed by both the debtor and the creditor that are subject to the violated commitment, which may require additional interaction between the agents.

\[ C(x, y, \text{CommitmentViolated}, \text{CreateCommitmentToCompensate}) \]

**Preference for Fulfillment** Although delegation and compensation provide intuitive mechanism to deal with infeasibility, in many situations it may not be possible to apply these approaches (e.g., the role may not have delegation power, compensation cannot be scheduled, etc.). In such situations it is inevitable for the agent to violate one or more of its commitments. Even in this case, it is important to decide which commitments to fulfill and which ones to violate. This decision can be made by taking different factors into account. For example, consider an agent that has two commitments to two different creditors and has to violate one of these commitments because of infeasibility. If the agent would care about the opinion of one creditor more than the other, it would be reasonable for the agent to violate the commitment of the less cared creditor. On the other hand, if there would be many number of infeasible commitments and violation of a single commitment, which requires the most resources, would resolve the infeasibility, then it would be reasonable to violate that commitment instead of violating several commitments with less resource requirements.

In order to deal with such situations, if it is possible at design-time, a role specification may include a preference relation over the commitments which defines a partial or total order to indicate which commitments to fulfill and which ones to violate in the case of infeasibility.

**Definition 9.** Given two commitments \( c_i \) and \( c_j \), \( c_i \triangleright c_j \) states that \( c_i \) is preferred for fulfillment over \( c_j \).

For instance, considering the two following commitments of the professor role, the commitment to attend a conference may be preferred (based on importance from the point of the role) for fulfillment over the fulfillment of the commitment to give a lecture.

\[ c_2 = C(\text{Professor, Department, PaperAccepted, ConferenceAttended}) \]

\[ c_3 = C(\text{Professor, Department, LectureScheduled, LectureGiven}) \]

If this is the case, then the professor role specification should involve the preference order \( c_2 \triangleright c_3 \) to state this situation. On the other hand, when it is not possible to define such a preference relation at design-time, then the agent itself should decide on which commitments to fulfill and which ones to violate in the case of infeasibility at run-time using other reasoning mechanisms.
5 Discussion

The abstraction of roles has been important in designing organization-based multiagent systems. However, understanding what goes into a role and how roles affect each other is an ongoing research question. This paper illustrates various problems that can arise in designing roles and suggests ways to overcome them. Once roles are conflict free, the next step is to study how the agents will enact these roles. To this end, various existing work studies, whether agents that enact these roles are fit to do so [3]. While this is important, there is not sufficient work on how an agent can stop enacting a role, how this would affect the agent’s existing commitments, and so on. Intuitively, the agent can be released from its conditional commitments. With active commitments, the problem is more tricky. For instance, assume that a review period is over before a program committee member fulfills its commitment of reviewing a paper. Would this commitment be considered violated even if the role no longer exists and thus binds the agent?

Commitment conflicts and norm conflicts, which are closely related to commitment conflicts, are studied in multiagent research literature [9, 16, 20]. These studies focus on the formal definition of different conflict types, such as conflicts due to logical inconsistencies and temporal disputes. Here, we use a general notion of conflict based on the concurrent obligation and prohibition of the same condition via commitments. However, the points we emphasize here such as the conflict-free role specifications and mutually exclusive roles are mainly independent from the specific definition of conflict. Hence, these points should be taken into account while designing any role based multiagent system.

In this paper we adapt a generic definition of commitment feasibility from our previous work [10]. Computing an agent’s commitments feasibility in practice is a challenging problem which requires to consider various resources of the agents, temporal constraints and also expectations of the agent in the interactions with others. In our previous work we use constraint satisfaction methods in order to achieve this objective. Here, we do not consider how feasibility of the agents commitments are computed. This can be done in different ways taking the structure of the considered multiagent system into account. On the other hand, we focus on what agents can do once an infeasibility is detected. However, the approaches that we present here are not the only responses that can be given by the agent to deal with infeasibility. For example, once infeasibility is detected an agent may take action to acquire more resources to resolve infeasibility. We left the investigation of such approaches as future work.

There are a lot of work on specifying organizations. AGR [6], Gaia [24], Tropos [1] and OperA [5] are important examples. AGR model specifies an organization through agents, groups, and roles. The interactions are generally specified with conditional actions. Both Gaia, Tropos and OperA focus on design of organizations with various rich constructs, such as capabilities, goals, and so on. Thus, their focus is on a global view of how these constructs exist together. Here, our aim is not on the interplay between various organizational constructs but on the conflicts that can arise within and among roles in an organization. On the other hand, integration of the concepts we present in this work into these frameworks would be beneficial for these approaches. Accordingly, we aim to extend at least one of these frameworks by introducing especially the conflict concept into the framework in our future research.
Formalization of organization level constraints is studied by van Riemsdijk et al. [15] using linear temporal logic in the context of MOISE+ organizational modeling language [11]. An organizational constraint is a meta-constraint in the level of organization, which specifies how an organizational action such as enacting to a role restricts an agent’s behavior (e.g., agent should adopt obligations of the enacted role) or cardinality requirements on the number of agents that enact a given role. While enforcing those constraints are obviously important, they are different than the constraints we are interested in here. Our concern here is at a lower-level to ensure that the roles are designed correctly to begin with.

Fornara, Viganò and Colombetti use institutional actions to define semantic of commitments mapping messages, which manipulate the state of a commitment, to institutional actions [8]. Beside institutional actions, they also consider norms in the form of event driven rules, which fire under certain conditions and cause creation of commitments that correspond to the norms. However, they are not concerned about the possible conflicts between the norms or roles as we have done here.

Dastani, Dignum and Dignum model a role as a set of goals and plans [3]. Obligations and prohibitions in the form of norms conceptually exist, but they are basically used to generate goals to be achieved and goals to be avoided, respectively. These type of conflicts are closely related to our definition of feasibility. However, our and their approaches consider this issue from different perspectives. In their perspective the conflict occurs with respect to a specific plan. On the other hand, in our perspective feasibility is independent from a specific plan or execution. Instead we use it as an indication of possible fulfillment and violation of agents’ commitments.

Odegard et al. discuss temporal aspects of roles [14], where they consider active and suspended states of roles. To capture the transitions between these states (and also for creation and termination) they define a set of operations. The first operation is classify, which is used to occupy a to an agent. The complementing operation declassify is used to make the role unoccupied again. Activate and suspend operation are used for the transitions from suspended to active and active to suspended states, respectively. In this paper, we use a simplified model through making roles active and passive based on some preconditions. This simplification is enough in our case, since we consider conflicts only when roles are active. However, it is straightforward to integrate our definitions into a more complete model of roles with more states.

Telang and Singh extend the TROPOS methodology [1] with commitments to model agent interaction [18]. In order to identify the commitments during the development, they first determine the major roles and goals of these roles. These high-level goals are decomposed into more fine-grained sub-goals. Finally, the goals are mapped to tasks that are required be performed by the agents who enact the roles to achieve the goals. They use these tasks and their dependencies to identify the commitments required to model the interaction in the system. Our approach can be coupled with theirs as a post-processing capability to check if the resulting roles are conflict-free.

References

Alternatives to Threshold-Based Desire Selection in Bayesian BDI Agents

Bernardo Luz¹, Felipe Meneguzzi², and Rosa Vicari¹

¹ Informatics Institute, Federal University of Rio Grande do Sul
Av. Bento Gonçalves, 9500 – Porto Alegre, Brazil
{bernardo.luz,rosa}@inf.ufrgs.br,
² School of Computer Science, Pontifical Catholic University of Rio Grande do Sul
Av. Ipiranga, 6681 – Porto Alegre, Brazil
felipe.meneguzzi@pucrs.br

Abstract. Bayesian BDI agents employ bayesian networks to represent uncertain knowledge within an agent’s beliefs. Although such models allow a richer belief representation, current models of bayesian BDI agents employ a rather limited strategy for desire selection, namely one based on threshold values on belief probability. Consequently, such an approach precludes an agent from selecting desires conditioned on beliefs with probabilities below a certain threshold, even if those desires could be achieved if they had been selected. To address this limitation, we develop three alternative approaches to desire selection under uncertainty. We show how these approaches allow an agent to sometimes select desires whose belief conditions have very low probabilities and discuss experimental scenarios.

1 Introduction

Due to its computable representation of practical reasoning and its folk psychological abstraction to autonomous reasoning, the beliefs, desires and intentions (BDI) model has been extensively studied within the autonomous agents community. Most traditional implementations of BDI agents include a logic-based belief base representing the knowledge an agent has about the world, and plan library that can be selected by an agent when it adopts certain desires. Once a course of action is selected by an agent for execution, it becomes part of an agent’s intention to which an agent commits to execute.

Beliefs are traditionally represented by a closed set of ground atomic literals, each of which is associated with a truth value, and consequently does not normally represent uncertainty. Nevertheless, there exist logical formalisms to represent uncertainty regarding an agent’s beliefs. Bayesian networks [9] are a popular way of representing uncertain information probabilistically, where parts of it are conditioned on others (e.g., cause and consequence relationships, diseases and symptoms). They are directed acyclic graphs (DAGs), whose nodes...
represent event variables associated with two or more possible states, and each state has an explicit occurrence probability.

Given the lack of support for uncertainty in the BDI agent model and the representational power of bayesian networks, there has been work focused on extending the BDI agent model to reason with uncertainty using bayesian networks [4]. This type of agent model no longer relies solely on ground literals to represent an agent’s belief base, but rather on a bayesian network.

Traditional BDI agents select desires (or plans with implicit desires) based on a binary condition on the literals of the belief base, under the assumption that this condition is minimal for the desire’s viability. The underlying idea is that, if the context condition is not true, then a desire and its associated plans have no chance of being successful. However, even if the context condition is true, a desire might be impossible and an intention associated with it might fail. Similarly, bayesian BDI agents are susceptible to selecting desires that cannot be satisfied in the current world state. Previous approaches to bayesian BDI reasoning [4] have relied on performing desire selection validation by applying a threshold on the probability being evaluated (e.g., that of the desire itself), so that if a certain logical query is less probable than the threshold, then the desire does not meet the minimal requirement to being successful. However, in a probabilistic world, context conditions are less crisply defined. In response, we have developed three alternative desire selection strategies that relax the requirement on the probability threshold for the context condition, and analyze situations where these strategies might be advantageous.

This paper is organized as follows: Section 2.1 presents BDI agents, Section 2.2 presents bayesian networks, Section 2.3 presents bayesian BDI agents, Section 3 presents bayesian BDI reasoning, Section 3.1 presents a threshold-based desire selection process, Section 4 presents alternative approaches to bayesian BDI desire selection, Section 4.1 presents Probability Ranking desire selection, Section 4.2 presents Biased Lottery desire selection, Section 4.3 presents Multi-Desire Biased Random Selection, Section 5 presents an example, and Section 6 presents our final considerations.

2 Background

In this section, we review previous efforts upon which our work is based. We start by briefly explaining the BDI model, then proceed to introducing the basics of bayesian networks and finally we enumerate existing work on bayesian BDI agents.

2.1 BDI Agents

Autonomous agents are often defined as encapsulated computer systems situated in an environment and capable of flexible autonomous action in this environment in order to achieve certain goals [5]. The agent must adapt itself to a dynamic environment, while seeking to fulfill its goals. In order to provide a stronger
computational grounding for this notion of agent, many architectures have been proposed, among which, one of the most widely studied is the one centered around the mental attitudes of beliefs, desires and intentions (or BDI) [2]. This architecture was originally proposed as a philosophical model of human \textit{practical reasoning}, that is, reasoning aimed at deciding how to act in the world towards achieving one’s goals.

Beliefs contain a representation, internal to the agent, of environment elements considered relevant for the agent’s reasoning. The state of an agent’s beliefs may contain either less information than the current state of the environment (e.g., because of limited sensing ability), or more (e.g., if the agent does additional information processing on its sensing). Desires represent objectives that the agent would like to achieve (i.e., they can be considered an agent’s \textit{motivation} [10]). Intentions are those desires that the agent has committed itself to achieving, as well as the steps towards achieving these desires. Agents resist abandoning their intentions, and, should a plan fail, it is often the case that they choose to re-plan.

A BDI agent selects desires through a process that considers the current viability and the absence of conflict with existing intentions. Desires often have preconditioning beliefs that indicate whether or not they should be selected by the agent, as a matter of logical evaluation [8].

\section*{2.2 Bayesian Networks}

Traditional first-order logic approaches to knowledge representation are insufficient to represent certain domains where there is uncertainty in the validity of statements over time [6, 11]. Examples of reasons for this limitation are the high cost of exhaustively representing all possible combinations of truth values using logic rules (laziness), the lack of a complete theory of the domain in question (theoretical ignorance), and the potential impossibility or inviability of performing all necessary tests to ascertain complete truth for certain statements (practical ignorance).

The fact remains that people commonly reason with incomplete knowledge and make decisions based on assumptions over unknown facts. This knowledge comprises what is \textit{known} to be true, what is \textit{not known} and \textit{estimates} based on relationships between elements of the world. Pearl [9] devised a formalism to represent partial knowledge based on the causal relationships between elements in the world, using probability theory to represent how knowledge about one element in the world influences the certainty about others related to it. Here, relationships between elements are represented in a network, and probabilities between related elements are calculated using \textit{Bayes’ Rule}, with the resulting formalism being called a \textit{Bayesian Network}. A bayesian network is a type of \textit{causal network} that allows the specification of knowledge where parts of it are conditioned on others, supporting the update of probabilities when new information (i.e., \textit{evidence}) is obtained.

Given two events $A$ and $B$, if we know the probability of $A$ given $B$ and the probability of $B$, we can calculate the probability of seeing both $A$ and $B$, \[ P(A \cap B) = P(A|B)P(B) \]
as shown in Equation 1, which represents the fundamental rule for probability calculus. It can also be conditioned on another event \( C \), as shown in Equation 2.

\[
P(A|B)P(B) = P(A \cap B). \tag{1}
\]

\[
P(A|B \cap C)P(B|C) = P(A \cap B|C). \tag{2}
\]

Equation 3 is the key equation behind bayesian networks: Bayes’ Rule. Bayes’ Rule makes it possible to update beliefs about an event \( A \), provided that we get information about another event \( B \). Thus, \( P(A) \) is usually called the prior probability of \( A \), whereas \( P(A|B) \) is called the posterior probability of \( A \) given \( B \). There is also a general version of Bayes’ Rule, in a context \( C \) – exhibited as Equation 4.

\[
P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{3}
\]

\[
P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)}. \tag{4}
\]

There may be evidence that a given variable is in a certain state. When this happens, it is said that such a variable is instantiated. This kind of evidence is called hard evidence. Conversely, if a statement about a variable state is made based on dependencies rather than explicit knowledge, it is said that there is soft evidence about that variable.

The \( d \)-separation property tells us if two variables are independent of each other in the current state of the bayesian network. There are three types of connection in the topology of a bayesian network: serial, diverging and converging. Each connection type accounts for a specific reasoning as to whether variables are \( d \)-separated or \( d \)-connected (what we call variables that are not \( d \)-separated)\(^4\).

In a serial connection, if we have no hard evidence about a variable, evidence about its parent/child passes through it, affecting our beliefs about it and about its uninstantiated child/parent. In a diverging connection, if we have no hard evidence concerning the parent, evidence about one of its children affects our beliefs about the other – uninstantiated – children. In a converging connection, if we have no hard evidence about the child or one of its descendants, evidence about a parent does not influence our beliefs about the other(s).

### 2.3 Bayesian BDI Agents

Although traditional implementations of BDI agents use a logic-based approach to model the world, these approaches fail to account for the uncertainty inherently associated with the real world. In order to address this shortcoming, work has been carried out to switch from a purely logical view of the agent’s beliefs

---

\(^3\) The equations in Section 2.2 have been extracted from [6].

\(^4\) “\(d\)” is for “directed graph”.

---

Page 211
based on traditional logic to a bayesian network, integrating it into the reasoning process of BDI agents. Fagundes et al. [4] have created an ontology-based BDI agent in which the belief base is replaced by a bayesian network and the desire selection process relies on probability thresholds to adopt new desires. Kieling and Vicari [7] integrate a bayesian network into an implementation of the Jason [1] AgentSpeak(L) [10] interpreter. Finally, Carrera and Iglesias [3] focus on the process of updating beliefs within a bayesian BDI agent.

3 Bayesian BDI reasoning

In this section, we develop a reasoning cycle that should be general enough that it could be used to describe the reasoning performed by previous work on bayesian BDI agents [3, 4, 7]. Later, we describe a desire selection process that is built around a threshold evaluation, as in [4], within such a reasoning cycle.

In this paper, we consider the belief base to correspond to an entire bayesian network whereby the causal relations between beliefs are explicitly represented. Moreover, given current evidence, we also explicitly represent the probability that a certain variable is in a particular state. Each event variable has $n$ possible states, each with an associated probability, that either is readily available from a conditional probability table if the state of all the parent variables is known (i.e., there is hard evidence on each of them) or has to be calculated.

Desires in the bayesian BDI agent model refer to specific event variable states in the bayesian network, and each desire has a preconditioning belief, indicating when that desire can be adopted by the agent. Our choice of belief-preconditioned desire representation follows the tradition of many implemented BDI systems (e.g., [8, 10]). We present two types of desire for bayesian BDI agents: strong and weak. Strong desires are desires on which there must be hard evidence so that they can be considered fulfilled. There may not be any doubt, however small, on whether or not a strong desire has been satisfied. Weak desires are those that are not necessarily expected to be confirmed via hard evidence, but are expected to be believed to be sufficiently likely to be true, i.e., to reach a certain minimum probability value. These may be viewed as desires that accept soft evidence as sufficient in order to be considered satisfied. Strong desires may be viewed as a special case of what would otherwise be weak desires, where, given each desire $d$, $P(d) = 1$.

Similarly to traditional BDI agents, intentions are desires to the fulfillment of which the agent has committed itself. The agent will seek a plan – a sequence of actions – that is applicable to the current situation: any plan that is aimed at satisfying at least one of the intentions and whose preconditions are not conclusively denied (i.e., preconditions that are not contradictory to hard evidence) is valid.

Algorithm 1 outlines a generic reasoning cycle for a bayesian BDI agent reasoning within an uncertain environment. First, the agent updates its belief base (i.e., the probabilities in the bayesian network), according to the latest perceptions from the environment (Line 2). The agent then proceeds to evaluate
Algorithm 1 Reasoning Cycle for Bayesian BDI Agents

1: procedure Reasoning Cycle for Bayesian BDI Agents 
2:   update beliefs based on percept 
3:   evaluate and possibly choose desires 
4:   seek plans that might satisfy the chosen desires 
5:   for each chosen desire, if an applicable plan has been found, create an intention 
   and associate it with the desire and the plan, which is therefore adopted 
6:   if a plan was not found, mark the desire as unsatisfiable at this time 
7:   if adopted plan failed, either seek another plan or remove the intention (subject 
   to commitment policy) 
8:   if adopted plan succeeded, desire is considered fulfilled and this is reflected on 
   beliefs 
9: end procedure

its possible desires (Line 3), if a desire was selected, the agent must commit 
to satisfying it by adopting an intention. Once it has committed it seeks plans 
capable of satisfying it (Line 4) and then executes the plans in an attempt to 
achieve the goal (Line 5). For each chosen desire for which no way of attempting 
to fulfill it has been found, mark it as “unsatisfiable at this time” and refrain 
from creating an intention for it in the current cycle (Line 6). If there is an 
adopted plan and it fails, then the agent may either seek an alternate plan or 
give up on the corresponding intention altogether (Line 7). This is subject to 
a commitment policy that may take into account whether this happened before 
to the desire associated with this intention, to intentions in general (there could 
conceivably be some kind of overall environment issue behind the failures), the 
rate at which alternate plans have proven effective, the computational cost for 
obtaining such plans – perhaps compared to the cost of desire selection, etc. 
Successful plan executions cause their associated intentions and, in turn, desires 
to be marked as satisfied (Line 8). This fact will be implicitly reflected in the 
agent’s beliefs in the next perception cycle, unless there is a contradicting change 
before then, e.g., by sensing an environment change caused by another agent. 

Since Bayesian BDI agents’ beliefs are extended with probabilistic data, it 
is no longer sufficient to perform the logical evaluation for each desire’s pre-
conditions to determine those that are eligible for intention creation. Just as 
there are degrees of probability in the beliefs, selecting a desire is now a deci-
sion made with varying degrees of confidence, which implies that preconditions 
are no longer strictly about validity of selection, but also about confidence in 
a selection that is made under uncertainty. The only case where it is a matter 
of validity is when there is hard evidence against the desire’s precondition (i.e., 
evidence of a different state of the event variable).

Moser et al. [4] performs reasoning using a threshold-based evaluation: if the 
probability being evaluated is equal to or greater than the threshold value, the 
associated event variable state is considered valid; in that work, the existence of 
a belief is dependent on this validation. Such reasoning involves checking if the 
probability associated with the applicability of the desire satisfies the threshold;
if so, the desire may be selected; otherwise, the agent simulates hard evidence on all combinations of the preconditioning event variable states – one state per variable – to determine which such state combinations would, if supported by hard evidence, allow for a threshold-satisfying probability of the event variable state corresponding to the desire itself, if any. New desires are then created from these states, connected to the original desire through causality.

Desires preconditioned on beliefs holding a probability that is exactly equal to zero, i.e., as a result of hard evidence on a state other than the one referred to by the belief in question, but associated with the same event variable, must not be selected. It is important to point out that once an intention has been dropped (i.e., not fulfilled), the desire is added back to the list of desires; without this, failed desires would be lost. Although this is not shown in any of the selection algorithms in this paper, as it is not a part of desire selection itself, it is an underlying assumption of the reasoning cycle.

3.1 Threshold-Based Desire Selection

In this section we describe a desire selection process that is threshold-based, which is a key characteristic in previous work [4], in terms of our reasoning cycle. This process is summarized in the pseudocode of Algorithm 2, representing a threshold-based desire selection algorithm in the context of a Bayesian BDI agent. It takes as parameters the numeric threshold value and the list of available desires (Line 1). The algorithm traverses the list of desires (Line 2) and, for each one, evaluates whether the probability of its associated precondition is greater than, or equal to the threshold (Line 3). If so, the desire in question is removed from the list of desires (Line 4) and returned (Line 5), thereby refraining from continuing to traverse the list, the implication being that this algorithm only selects one desire. If the entire list of desires is traversed and no desire has been selected (Line 7), the algorithm returns null (Line 8), denoting that no new desire is pursued by the agent.

Algorithm 2 threshold-based selection

1: function ThresholdBasedSelection(threshold, desires)
2: for each desire such that desire ∈ desires do
3: if desire.preCondition.probability ≥ threshold then
4: desires.remove(desire)
5: return desire
6: end if
7: end for
8: return null
9: end function
4 Alternatives for Bayesian BDI Desire Selection

The threshold-based desire selection algorithm shown in Section 3 avoids selecting desires whose belief preconditions do not meet a minimal degree of probabilistic support. As such, it constitutes a relatively simplistic mechanism for desire selection in a probabilistic section, and suffers from two key limitations. On the one hand, as the selection threshold approaches one, the agent becomes extremely conservative, and may not select any desire and remain idle for long periods of time.

On the other hand, if the threshold approaches zero, the agent becomes less strict in ensuring the viability of the desires it chooses to pursue. Importantly, depending on the order in which the desires are checked, the agent might select desires that are less likely than others. Moser et al. [4] work with the notion of incompatible desires in Bayesian BDI agents, which is beyond the scope of this work. These incompatible desires are sorted by probability and the one with the highest probability is selected. The selection process for multiple desires that are not considered incompatible is not a concern in their work; there, competition is not assumed to be a part of the desire selection process. In this paper, we assume that desires do not conflict with each other, or that there is a process that filters conflicting desires. Moreover, we assume competition among desires during selection, unless otherwise specified.

In order to address the limitations of threshold-based selection, we propose a number of alternative desire selection mechanisms that ensure a finer control over an agent’s choice of desires while taking into consideration the probabilistic nature of an environment. These approaches eliminate idleness and ensure that more likely desires are selected more often. In the algorithms developed in this section, similarly to the desire selection algorithm shown in Section 3, we assume that once an intention is dropped the desire is added back to the list of desires.

4.1 Probability Ranking

This approach involves sorting the desire list in decreasing order of precondition probability, resulting in a ranking from highest to lowest probability precondition, and picking up the desire backed by the belief most likely to be true. The pseudocode in Algorithm 3 illustrates the Probability Ranking desire selection algorithm. Its only parameter is a list of desires (Line 1). If there are any desires (Line 2) the algorithm sorts them by precondition probability (rankedDesires, Line 3), selects the first desire (Line 4), removes it from the list (Line 5) and, if the probability of that desire’s precondition is greater than 0 (Line 6) – to prevent a desire associated with a contradicted precondition from being selected – returns that desire (Line 7). Otherwise, the algorithm returns null (Line 10).

4.2 Biased Lottery

Selecting desires by ranking them over their precondition probability as we show in Section 4.1 helps ensure that an agent is never idle. However, it is still possible that certain desires will never be selected, even if they were possible but
Algorithm 3 Probability Ranking Selection

1: function ProbabilityRankingSelection(desires)
2:     if desires.length > 0 then
3:         rankedDesires := desires ordered by precondition probability
4:         desire := rankedDesires.first()
5:         desires.remove(desire)
6:         if desire.preCondition.probability > 0 then
7:             return desire
8:         end if
9:     end if
10:    return null
11: end function

were weakly supported by the agent’s beliefs. Situations where this is detrimental to the agent occur when the agent has not obtained enough evidence about the environment, or has obtained the wrong evidence. In order to address that limitation, we now develop a technique that randomly picks desires using their precondition probability to weight this selection. The idea is to randomly generate a number and use it to determine which desire to choose, according to a probability distribution reflecting the probabilities of the desires’ preconditions.

In order to generate this probability distribution over the desires, we generate a series of numeric intervals within the [0, 1) range assigning, for each belief, an interval proportional to the probability of their belief precondition. The probabilities, thus, serve as weights that create bias in what would otherwise constitute a purely random selection; it is a nondeterministic desire selection that is subject to bias from the precondition probability. This desire selection method neither disregards desires backed by beliefs holding very low probabilities, nor is designed to embrace them more often than common sense would permit – than such probabilities would suggest. We formalize this selection mechanism in the pseudocode of Algorithm 4, which uses the function described in Algorithm 5 to generate the selection probability intervals. Algorithm 4 takes as input the list of desires (Line 1) and generates a random numeric value (Line 2) and a list of numeric values (Line 3) that correspond to the upper limits (boundaries) for the numeric intervals used in desire selection; Function GenerateIntervals (Line 3) is detailed in Algorithm 5. The algorithm proceeds to traverse the list of upper interval limits (Lines 4–10); it uses the randomly generated number to select a desire (Lines 5 and 6), which is then removed from the list of desires and returned (Lines 7 and 8). If the entire list of upper interval limits is traversed and the random value has not been found to belong to any of the intervals (Line 10), the algorithm returns null (Line 11).

Algorithm 5 takes as input a list of desires (Line 1) and starts by creating a list to store the upper numeric interval limits that will be calculated (intervals, Line 2). Provided that there are elements in the input list, the algorithm proceeds to create a list that will contain the probabilities of the desires’ preconditions (Lines 3 and 4). It also defines a variable sum that will be used to store the
Algorithm 4: Biased Lottery

1: function BiasedLottery(desires)
2:     randomValue := random number ∈ [0, 1]
3:     intervals := GenerateIntervals(desires)
4:     for i = 0 to intervals.length do
5:         if randomValue < intervals[i] then
6:             desire := desires[i]
7:             desires.remove(desire)
8:             return desire
9:     end if
10: end for
11: return null
12: end function

The sum of all such probabilities, initializing it to 0 (Line 5). It then traverses the
desire list (Lines 6–9) storing the probabilities of the desires’ preconditions in
the corresponding positions of probabilities and accumulating the probability
of all desire preconditions in the sum variable (Lines 7 and 8). If the sum is
greater than 1 (Line 10), it normalizes the probabilities and uses these values as
interval sizes while generating numeric intervals (Lines 12–14). If not (Line 15), it
generates numeric intervals using the probabilities as interval sizes (Lines 17–19).
Lines 11 and 13 are the normalized equivalents of Lines 16 and 18, calculating
and ultimately assigning upper interval limits to the positions in intervals.

We do not perform normalization when the sum of the precondition probabil-
ities is less than 1.0, as this would inflate selection probabilities for desires pre-
conditioned on insignificant events. For example, a single desire preconditioned
on a belief with 0.0001 probability would be treated as though its probability
were 1.0. Note that the numeric intervals for the desires are forced not to inter-
sect with one another, since the one randomly generated number (per selection
cycle) is expected to select, at most, one desire-associated numeric interval. Al-
though this algorithm now allows an agent to sometimes pick desires that would
not normally be selected, it is still limited to the choice of a single desire.

4.3 Multi-Desire Biased Random Selection

This approach to desire selection removes inter-desire competition, by consider-
ing desires independently of each other (e.g., full parallelism is possible), allowing
multiple desires to be selected simultaneously. Given a desire \( D_i \) preconditioned
on a belief holding a probability \( P_i \), we say that \( D_i \) is assigned a numeric in-
terval \( I_i = [0, P_i] \). This is done for every pending (i.e., unfulfilled) desire. For
every such desire \( D_i \), if a randomly generated numeric value \( N_i \) in interval \([0, 1]\)
belongs to interval \( I_i \), the desire is added to the set of desires to be selected at
the end of this selection cycle.

The pseudocode of Algorithm 6 formalizes our proposed approach for Multi-
Desire Biased Random Selection. It takes as input the list of desires (Line 1), and
Algorithm 5 Biased Lottery – Desire Intervals

1: function BiasedLottery:GenerateIntervals(desires)
2:     intervals[desires.length] 
3:     if desires.length > 0 then 
4:         probabilities[desires.length] 
5:         sum := 0 
6:         for i := 0 to desires.length do 
7:             probabilities[i] := desires[i].preCondition.probability 
8:             sum := sum + probabilities[i] 
9:         end for 
10:        if sum > 1 then 
11:            intervals[0] := \frac{probabilities[0]}{sum} 
12:            for i := 1 to intervals.length do 
13:                intervals[i] := intervals[i − 1] + \frac{probabilities[i]}{sum} 
14:            end for 
15:        else 
16:            intervals[0] := probabilities[0] 
17:            for i := 1 to intervals.length do 
18:                intervals[i] := intervals[i − 1] + probabilities[i] 
19:            end for 
20:        end if 
21:    end if 
22:    return intervals 
23: end function

creates a list that will be used to store any number of desires that may be selected (selectedDesires, Line 2). This selection takes place by traversing the list of desires (Lines 3–8), and randomly picking desires based on their precondition probability (Lines 4–6).

5 Example

In order to illustrate the effects of each desire selection strategy described in Section 4, we now introduce a working example to show how an agent would react to situations using our proposed algorithms. Our example scenario consists of a watchman agent that is tasked with guarding an installation. The presence of suspicious people nearby increases its estimate of a security breach. There is an alarm in the installation, that is effective under normal circumstances. However, there are reports of occasional electrical malfunctions in the installation, which may cause the alarm to ring for no reason or not to ring when it is expected to. Moreover, the watchman becomes interested in seeking evidence that there is not an electrical malfunction if it knows that there are suspicious people nearby. The surrounding area is known for intense traffic, and accidents are more common than in most other areas, resulting in noise that is almost always perceived by the agent. However, noise might be caused by trespassers, though that is not very likely. In order to patrol the installation, the watchman periodically chooses...
between the default and an alternate route, and it becomes more inclined to patrol the alternate route as its belief that a security breach is either imminent or already taking place increases, and conversely, the watchman is more inclined to patrol the default route when everything looks calm.

Regarding the relationships among the event variables in the network, we note that: i) Evidence of the presence of suspicious people nearby increases the probability of a security breach; ii) Evidence of the alarm activating increases the probability of a security breach occurring, as well as the probability of there being suspicious people nearby. This is still true if there is also evidence of an electrical malfunction, but the probability increase for both event variable states is smaller. If there is evidence that there is no electrical malfunction (e.g., a notification about maintenance very recently performed), the probability increase is the greatest of the three cases; iii) evidence of noise increases the probability of a security breach. However, this increase is almost nullified upon evidence of an accident, as this network tells us that an accident is a much more probable cause of noise than a security breach, and the impact of a security breach on the probability of noise if we already know of an accident is small; and iv) An increase on the probability of a security breach (e.g., through evidence of suspicious people and noise) increases the probability of the alarm activating, even if there is an electrical malfunction, though then the probability increase is smaller.

The bayesian belief base of the watchman encoding the domain knowledge described in the scenario is represented in Figure 1. Do note that we do not associate the Route variable with a belief about the environment state, but rather we associate it with an internal belief associated with the agent’s currently chosen route. It is not a part of the reasoning surrounding the probability of a security breach or any of the other event variables, and this is the reason we left it disconnected from all the other nodes in the network.

This watchman agent has two mutually exclusive strong desires that are periodically renewed: 5 Route.default(SecurityBreach.false) and Route.alternate(

---

5 We denote desires in the form <desire>(<preconditioning belief>), where both elements are described as <event variable>.<state>.
SecurityBreach.true). That is, the agent desires to patrol the default route if there has not been a security breach and the alternate one otherwise. It also has the weak desire ElectricalMalfunction.false(SuspiciousPeople.true), i.e., the desire to believe that there is no electrical malfunction at the moment, conditioned on the presence of suspicious people nearby, as well as the strong desire Accident.true(Noise.true), i.e., the desire to discover that there has been an accident, if noise has been heard.

We now briefly present the result of using each of the four algorithms while working with an initial scenario – where what happens during the execution of each algorithm is not carried over to the next – where there is no hard evidence of any event. Since, there is no hard evidence yet, \( P(\text{SuspiciousPeople}) = (0.3, 0.7) \) (i.e., \( P(\text{SuspiciousPeople} = \text{true}) = 0.7 \) and \( P(\text{SuspiciousPeople} = \text{false}) = 0.3 \)), and consequently \( P(\text{SecurityBreach}) = (0.073, 0.927) \); also, \( P(\text{Noise}) = (0.0624, 0.9376) \). Desire Route.default is preconditioned on a belief that SecurityBreach is false, which has a 0.927 probability; desire Route.alternate is preconditioned on a belief that SecurityBreach is true, which holds a 0.073 probability; desire ElectricalMalfunction.false is preconditioned on a belief that SuspiciousPeople is true, which holds a 0.7 probability; and desire Accident.true is preconditioned on a belief that Noise is true, which holds a 0.0624 probability.

First, let us consider threshold-based desire selection, with a threshold of 0.75. This means that only Route.default(SecurityBreach.false) is an eligible desire for selection, since \( P(\text{SecurityBreach} = \text{false}) = 0.927 \). In a scenario where there is hard evidence of noise (i.e., \( P(\text{Noise} = \text{true}) = 1 \)), the probability
of suspicious people nearby is increased: $P(\text{SuspiciousPeople} = \text{true}|\text{Noise} = \text{true}) = 0.7422$. However, desire $\text{ElectricalMalfunction.false(SuspiciousPeople.true)}$ still fails to satisfy our threshold even so ($0.7422 < 0.75$). A lower threshold would work in this case, but a precondition’s probability might always be smaller than the threshold, if evidence able to increase its probability enough is never obtained – this exemplifies how there may be desires that are never selected using this criterion. One might suggest simply lowering the threshold to an extremely low value, but without other criteria we would then simply have a desire selection process that is indifferent to the various probabilities presented.

If we use Probability Ranking desire selection, we get the following ranking:

1. $\text{Route.default(SecurityBreach.false)}$: 0.927
2. $\text{ElectricalMalfunction.false(SuspiciousPeople.true)}$: 0.7
3. $\text{Route.alternate(SecurityBreach.true)}$: 0.073
4. $\text{Accident.true(Noise.true)}$: 0.0624

The agent will desire to patrol the default route, then to establish that there is no electrical malfunction, then to patrol the alternate route, and finally to verify if there has been an accident, in this order, unless a belief update (e.g., evidence that $\text{SecurityBreach = true}$) causes the ranking to be modified. Note that although the preconditioning probabilities serve as a criterion for sorting the desires, the probability values by themselves have no impact on how often the desires may be selected, so $\text{Accident.true(Noise.true)}$ will be promptly selected in the absence of higher-ranked desires regardless of the fact that its precondition holds a low probability.

If we use Biased Lottery, we get the following list of numeric intervals for the desires (the order is irrelevant):

- $\text{Route.default(SecurityBreach.false)}$: [0.0, 0.526)
- $\text{Route.alternate(SecurityBreach.true)}$: [0.526, 0.5674)
- $\text{ElectricalMalfunction.false(SuspiciousPeople.true)}$: [0.5674, 0.9646)
- $\text{Accident.true(Noise.true)}$: [0.9646, 1.0]

The sum of the desires’ precondition probabilities is greater than 1, so these values are normalized in the $[0, 1]$ interval and used to generate the intervals. Following the algorithm, a numeric value in the $[0, 1]$ interval is generated, and whichever interval it belongs to determines which desire is selected – if there were not a normalization, it could also tell us that no desire should be selected, by not belonging to any of the intervals. In this example, desire $\text{Route.default(SecurityBreach.false)}$ has a 0.526 probability of being selected, desire $\text{Route.alternate(SecurityBreach.true)}$ has a 0.0414 probability of being selected, desire $\text{ElectricalMalfunction.false(SuspiciousPeople.true)}$ has a 0.3972 probability of being selected, and desire $\text{Accident.true(Noise.true)}$ has a 0.0354 probability of being selected, each one competing with the others. So, if the randomly generated number is 0.3 (and thus within the first interval), the agent performs a patrol through the default route, or if the random number is 0.55, the patrol is through the alternate route.
If we use Multi-Desire Biased Random Selection, we get the following list of numeric intervals for the desires (the order is irrelevant):

- \textit{Route.default}(\textit{SecurityBreach.false}): [0.0, 0.927]
- \textit{Route.alternate}(\textit{SecurityBreach.true}): [0.0, 0.073]
- \textit{ElectricalMalfunction.false}(\textit{SuspiciousPeople.true}): [0.0, 0.7]
- \textit{Accident.true}(\textit{Noise.true}): [0.0, 0.0624]

For each of the four desires a numeric value in the [0, 1] interval is generated, and if the numeric value belongs to the corresponding desire's numeric interval, the desire is selected. In this example, desires \textit{Route.default}(\textit{SecurityBreach.false}), \textit{Route.alternate}(\textit{SecurityBreach.true}), \textit{ElectricalMalfunction.false}(\textit{SuspiciousPeople.true}) and \textit{Accident.true}(\textit{Noise.true}) have, respectively, 0.927, 0.073, 0.7 and 0.0624 probabilities of being selected, whereas each possible selection is fully independent of the others, thus rendering the selection process passive to yield multiple selected desires.

6 Conclusions

From a conservative standpoint, one may argue that threshold-based selection is sensible as it is, as resources will not be used \textit{without justification}. However, we believe that ignoring desires that are \textit{probabilistically irrelevant} in desire selection is not necessarily a rational choice, since it precludes an agent from exploring an environment. In response, we have developed three desire selection strategies that try to overcome this limitation.

In Probability Ranking selection, desires that would be ignored by threshold-based selection do get a chance, though only after the ones that would be accepted by it. However, it might be undesirable to select a desire preconditioned on a belief holding a very low probability just because there is no better alternative.

In Biased Lottery, we rely on nondeterminism to consider all desires while ensuring that desires backed by beliefs holding high probabilities should be selected more often than those backed by beliefs holding low probabilities, in proportion to their probabilities. Ideally, the probability of selecting each desire would be the same as the one associated with its precondition. However, in the cases where the total sum of desire probabilities is greater than 1, the competition between the desires in question proportionally reduces the individual probabilities of selection for each desire.

In Multi-Desire Biased Random Selection, we also rely on nondeterminism for the same reason. A key difference is that the desires are considered independently of one another, so that there is no competition among the desires, thus the number of desires possibly selected is not limited to one. One limitation of our work is that we do not deal with the issue of desire incompatibility, as this would pose significant problems in a probabilistic setting.

The nondeterministic nature of Biased Lottery and Multi-Desire Biased Random Selection makes it so that the watchman agent’s behavior may not be anticipated by a third party (e.g., another agent) intent on exploiting it. Such an
exploitation could involve forging an accident to drive away suspicion arising from a noise heard by the watchman, for instance. Hiring an employee to plant false evidence of an electrical malfunction would also impact the watchman’s beliefs, as a second, albeit more roundabout method of attempting to manipulate the watchman. This is an agent trait that we now describe as unpredictable proactiveness: agent behavior at a specific point in time cannot be completely determined by analyzing its beliefs, and is thus resistant to exploitation by a third party. Finally, our future work aims to evaluate the algorithms developed in different scenarios, and to investigate joint uses of Biased Lottery and Multi-Desire Biased Random Selection while considering desire incompatibilities.

References

Deciding Between Conflicting Influences

Andreas Schmidt Jensen
Department of Applied Mathematics and Computer Science, Technical University of Denmark, Matematiktorvet, Building 303B, DK-2800 Kongens Lyngby, Denmark

Abstract. This paper investigates an approach of decision making internally in an agent in which a decision is based on both preference and expectation. The approach uses a logic for qualitative decision theory proposed by Boutilier in order to express such notions. To make readily use of this we describe a simple method for generating preference and expectation models that respect certain rules provided by the agents.

1 Introduction

Agents taking part in a multi-agent system are usually seen as intelligent entities that autonomously are able to bring about (from their own perspectives) desirable states. In a fixed setting with a controlled number of agents and globally desirable states, the designer will in many cases be able to implement the agents such that their own desirable states coincide with the globally desirable states. In open societies agents often come from different sources and their desires cannot as such be assumed to match the global desires. A suggestion is to impose an organization upon the agents which is able to influence the actions of the agent towards the desires of the organization.

When agents are constrained by an organization, their own goals may conflict with those of the organization. In some of the previous work towards resolving such conflicts the solution has often been to order desires and obligations a priori, such that an agent either prefers desires over obligations or obligations over desires. This results in agents that are always selfish or always social. We argue in this paper that such distinction can be too hard; even a selfish agent could in some cases benefit by preferring certain obligations over its desires. We consider an approach on how to resolve such conflicts which is based on work in the area of qualitative decision theory by Boutilier [3], where the expected consequences of bringing about a state are considered. We show that this result in agents that are not always either social or selfish, but are instead able to decide based on the consequences of bringing about the states.

In order to make the approach readily useful we furthermore describe a simple method for generating models for preference and expectation, based on basic rules specified by the agents, such as “I prefer to drive to work when it is raining”. This will help building agents that are able to reason using our approach.
The paper is organized as follows: In section 2 we discuss the issues that arise when an agent has to make a decision between conflicting influences. In section 3 we present a new approach on how to solve such conflicts without having to put the agents into the categories “selfish” or “social”. We present a method for generating models that conform with the agent’s preferences in section 4. In section 5 we discuss a case in which agents have conflicting desires and show that our method enables them to choose using both their own preference and the expected consequence of bringing about each influence. Finally we conclude our work and discuss future research directions in section 6.

2 Conflicting influences

Agents entering an environment will be subject to influences from multiple sources; their own desires, requests from others, and obligations from an organization. In the well-known BDI model an agent’s desires become intentions, when the agent commits to bringing about these desires. One could argue that if an agent wants to accept requests from other agents, or if it wants to adhere to the obligations of an organization, these influences are merely desires as well; the agent simply desires to do so. The incentives for doing so are however not clear, since there should be different reasons for committing to actual desires and to requests or obligations disguised as desires. For example, if an agent has a desire to move a box from A to B, it typically wants to do so. However if the agent wants to pay a bill before its due date, this “desire” has more likely arisen from the fact that the agent does not want to pay a fine, rather than being an actual desire to pay the bill. In such a situation the desire may actually be an obligation or a request to pay the bill, which means that the agent should reason differently, since the actual desire is to avoid paying a fine.

Furthermore, consider an agent that receives an undesirable request from an agent that it desires to help. It may choose to commit to the task even though the task itself is not desirable, since the desire to help the other agent is stronger than the desire to not perform the task (the consequence of not helping the other agent might be a bad reputation). Similarly if an agent is obligated to perform certain tasks for an organization, it should not only be able to consider whether the task is desirable, but also weigh this against the penalty for violating the obligation.

In this paper we call all the propositions that the agent has to take into account when making a decision “decision influences” since they influence how the agent chooses to act. It naturally has to take into account its desires, since it would be irrational to ignore them, but the consequence of not reasoning about e.g. obligations might be intolerable so these influence the agent as well. This also means that the agent is not supposed to be reasoning explicitly about whether it should commit to bringing about an arbitrary obligation or desires, since they are merely considered influences. Several approaches are proposed on how to let agents choose between such specific influences [1,4,5,6], so we briefly discuss how our solution differs.
In [4] conflicts between beliefs, obligations, intentions and desires are discussed, with a focus on a distinction between internal conflicts, e.g. contradictory beliefs, conflicting obligations and external conflicts, such as a desire which is in conflict with an obligation. The solution proposed, the BOID architecture, impose a strict ordering between beliefs, obligations, intentions and desires, such that the order of derivation determines the agent’s attitude. Thus different agent types emerge; an agent deriving desires before beliefs is a wishful thinking agent, while an agent deriving obligations before desires is social.

We believe this ordering is too strong; if an agent is social it will always choose obligations over desires, and vice versa for selfish agents. This might not always be appropriate. For instance, a selfish agent might desire not to go to work, but if the consequence of not fulfilling the obligation of going to work is severe (i.e. getting fired), even a selfish agent should consider this consequence before deciding not to go to work.

Dignum et al. suggests that “both norms and obligations should be explicitly used as influences on an agent’s behavior” [5]. In their approach obligations (and norms) are represented using Prohairetic Deontic Logic [8], a preference-based dyadic deontic logic which allows for contrary-to-duty obligations (obligations holding in a sub-ideal context). Furthermore they propose modified BDI- interpreter in which selected events are augmented with potential deontic events which, put simply, are obligations and norms that may become applicable when a plan is chosen. For instance, if agent a has an obligation to perform a task for another agent b, and a does not intend to do so he ought to inform b about this. The modified interpreter generates a number of options depending on these potential events and a relevant plan is chosen based on the agent’s attitude.

In [6] it is argued that the preference orderings induced by desires, obligations and norms should be combined into a single ordering. It is noted that a common way to do so is to allow that a single preference ordering determine the aggregate ordering, such that the agent might always put obligations over norms and norms over desires, similarly to the BOID architecture. Another approach is also discussed in which the orderings are mapped into a common scale, such that very desirable situations could outweigh the cost of violating certain obligations. Such ordering should be quite dynamic since, as noted, obligations towards a trusted agent should become less important if that agent becomes less trustworthy. Simple rules are presented to deal with few alternatives, but it is noted that the situation is more complex if an agent has to choose between three or more alternatives and none of the three orderings agree on a preferred alternative. A simple rule which orders the alternatives in a fixed order results in a very simple-minded agent and it is suggested that the consequences of different situations is considered, however this is not investigated further.

The approach presented in this paper compares the influences by taking the agent’s preferences into account such that more preferred influences will be chosen over less preferred influences. When influences are equally preferred there are two possibilities, either let the agent choose arbitrarily between the influences
or compare them using a different approach. When considering only desires the former option might be acceptable since if the agent wants to do two things and one is not more preferred than the other, it might seem reasonable to choose one arbitrarily. On the other hand, when influences include those originating from other agents or an organization the latter solution is more suitable. Otherwise since only the agent’s preference is considered, no reasoning about violating obligations or ignoring requests occur.

We suggest that the agent should reason about the expected consequences of choosing to pursue a decision influence and furthermore that this reasoning should be based on tolerance. When two decision influences are equally preferable, the agent should consider whether the consequences of bringing about one influence are more tolerable than the consequences of bringing about the other. We define a proposition as being tolerated when its negation is not preferred (e.g., working is tolerated if staying at home is not preferred over working). The reason for using tolerance instead of preference in the case of consequences is that the agent should not need to desire the consequences of bringing about a state. The consequences are side effects which may not be desired in the same way as the actual influence is. Of course, if a consequence is preferable, then clearly it is also tolerable but the opposite need not be the case. This means that even though none of the expected consequences are actually preferable we are still able to compare them. Finally, if the expected consequences are equally tolerable then the agent is allowed to choose arbitrarily between the influences.

Note that our approach does not incorporate the notion of an organization as such; the focus is on the propositions that may influence the agent’s reasoning, such as the obligations toward an organization. As a result, we model consequences as expectations from the environment, that is, which possible world is the most expected, which is the second most and so on. This means that if the consequences of the violation of an obligation are specified in an organizational model, these consequences are in our approach modeled such that worlds, in which the violation has occurred and the consequences are in action are more expected than those where the violation has occurred without resulting in any consequences. This will be evident in the example in section 5 where all expected consequences are incorporated into the same model.

3 Modeling Influence and Consequence

We base our work on the Logic for Qualitative Decision Theory (QDT) by Boutilier [2,3] by extending the notion of preference to allow multiple modalities in order to represent the preference of individual agents. The semantics and axiomatization are those of QDT and we define a few new abbreviations to be used in the decision making.

The basic idea behind the QDT model is as follows. An agent has the ultimate desire of achieving the goals to which it is committed. This can be modeled by a possible worlds-model in which the agent has achieved its goal when it is in a world where those goals hold. The most preferred world in an ideal setting is...
the world in which all of the agent’s goals are achieved. However, such world is often unreachable, for a number of reasons: the agent could have contradicting goals, other agents could prevent the agent from achieving all of its goals, an organization could impose obligations which contradict the agent’s goals, etc. By ordering the worlds in a preference relation it is possible to choose the most preferred world(s) in a sub-ideal situation.

In our approach we require that the consequence of bringing about a state should be taken into account. If the consequence of pursuing a personal desire is to be fired from your workplace, it might not be reasonable to do so even though the desire was more preferred than the obligations from work. We briefly describe QDT below with the semantics to include the notion of multiple agents before moving on to modeling the expected consequence of bringing about a state.

A QDT model is of the form:

\[ M = \langle W, Ag, \leq_1, \ldots, \leq_n, \leq_N, \pi \rangle \]

where \( W \) is the non-empty set of worlds, \( Ag \) is the set of agents, \( \leq_i \) is the transitive, connected preference ordering for each agent\(^1\), \( \leq_N \) is the transitive, connected normality ordering, and \( \pi \) is the valuation function. The normality ordering is used to model how likely each world is, e.g. it is normally cold when it is snowing.

The semantics are given as follows:

\[
M, w \models p \iff p \in \pi(w) \\
M, w \models \lnot \varphi \iff M, w \not\models \varphi \\
M, w \models \varphi \land \psi \iff M, w \models \varphi \land M, w \models \psi \\
M, w \models \Diamond_i \varphi \iff \forall v \in W, v \leq_i w, M, v \models \varphi \\
M, w \models \Box_i \varphi \iff \forall v \in W, w <_i v, M, v \models \varphi \\
M, w \models \Box_N \varphi \iff \forall v \in W, v \leq_N w, M, v \models \varphi \\
M, w \models \Diamond_N \varphi \iff \forall v \in W, w <_N v, M, v \models \varphi
\]

We can define the other operators (\( \lor, \to, \Diamond, \Box \)) as usual. Finally we can talk about a formula being true in all worlds or some worlds: \( \Diamond_i \varphi \equiv \Box_i \varphi \land \Diamond_i \varphi \) and \( \Box_i \varphi \equiv \Diamond_i \varphi \lor \Box_i \varphi \), respectively (similarly for normality). The following abbreviations are defined:

\[
(1) \quad I(\psi | \varphi) \equiv \Box_i \varphi \lor \Diamond_i \varphi (\varphi \land \Box_i \varphi) \quad \text{(Conditional preference)}
\]

\[
(2) \quad \varphi \leq_i \psi \equiv \Diamond_i \varphi \rightarrow \Diamond_i \psi \quad \text{(Relative preference)}
\]

\[
(3) \quad T(\psi | \varphi) \equiv \neg I(\neg \psi | \varphi) \quad \text{(Conditional tolerance)}
\]

\[
(4) \quad \varphi \Rightarrow \psi \equiv \Box_N \varphi \lor \Box_N (\varphi \land \Box_N (\varphi \rightarrow \psi)) \quad \text{(Normative conditional)}
\]

\(^1\) We adopt the notion by Boutilier and others that we prefer minimal models, so \( v \leq_i w \) denotes that according to agent \( i \), \( v \) is at least as preferred as \( w \).
The abbreviations state that (1) \( \psi \) is ideally true if \( \phi \) is true, (2) \( \phi \) is at least as preferred as \( \psi \), (3) \( \psi \) is tolerable given \( \phi \) and (4) that \( \psi \) normally is the case when \( \phi \) is.

In order to make decisions as motivated above, we define the following abbreviations, which allow us to specify different kinds of relative preference, and relative tolerance.

\[
\begin{align*}
\phi \not\leq_P \psi &\equiv \neg (\phi \leq_P \psi) \quad \text{(Not as preferred)} \\
\phi <_P \psi &\equiv (\phi \leq_P \psi \land \psi \not\leq_P \phi) \quad \text{(Strictly preferred)} \\
\phi \approx_P \psi &\equiv (\phi \leq_P \psi \land \psi \leq_P \phi) \lor (\phi \not\leq_P \psi \land \psi \not\leq_P \phi) \quad \text{(Equally preferred)} \\
\phi \leq_T^{(\gamma)} \psi &\equiv (T(\phi \mid \gamma) \land \neg T(\psi \mid \gamma)) \lor (T(\phi \mid \gamma) \leftrightarrow T(\psi \mid \gamma)) \land \left( \phi \leq_P \psi \lor \phi \approx_P \psi \right) \quad \text{(Relative tolerance)}
\end{align*}
\]

Thus \( \phi \) is at least as tolerable as \( \psi \) w.r.t \( \gamma \) when either \( \phi \) is tolerable given \( \gamma \) and \( \psi \) is not, or both \( \phi \) and \( \psi \) are tolerable given \( \gamma \) (or both are not), and \( \phi \) is at least as preferred as \( \psi \), or they are equally preferable. This means that even if neither is tolerable, they are still comparable.

### 3.1 Making a decision

We now show how the extended QDT-logic can be used to decide between conflicting desires and obligations. We define a model for decision making as follows:

\[
\mathcal{M}_C = \langle M, F, C, B \rangle,
\]

where

- \( M \) is an extended QDT-model as defined above,
- \( F \) is for each agent the set of influences,
- \( C \) is for each agent the set of controllable propositions\(^2\),
- \( B \) is the belief base for each agent.

We define the set of potential consequences \( C'(i) \) for an agent \( i \) such that if \( \varphi \in C(i) \) then \( \varphi, \neg \varphi \in C'(i) \).

**Definition 1 (Expected consequences).** Given an agent \( i \), its belief base \( B(i) \) as a conjunction of literals, the set of potential consequences \( C'(i) \) and a literal \( \varphi \). The expected consequences of bringing about \( \varphi \), denoted \( EC_i(\varphi) \), is given by:

\[
EC_i(\varphi) = \bigwedge C_\varphi \text{ for all } C_\varphi \in \{ C_\varphi \mid (B(i) \land \varphi \Rightarrow C_\varphi) \text{ where } C_\varphi \in C'(i) \}
\]

i.e. the conjunction of all literals \( C_\varphi \) that are normally consequences of bringing about \( \varphi \) given the current belief base. If there are no expected consequences, then \( EC_i(\varphi) = \top \).

\(^2\)A controllable proposition is, roughly, a proposition which the agent is able to influence, directly or indirectly, by an action. E.g. snow is not controllable and cannot be a consequence of an action, whereas work is.
Consider an agent $i$, and a normality ordering in which we have that

$$A \land \alpha \Rightarrow B, \quad A \land \neg \alpha \Rightarrow C, \quad D \land \neg \alpha \Rightarrow E,$$

and belief base $B(i) = \{\alpha\}$. Then we have that $EC_i(A) = \{B\}$ and $EC_i(D) = \emptyset$. If $B(i) = \{\neg \alpha\}$, then $EC_i(A) = \{C\}$ and $EC_i(D) = \{E\}$.

An agent $i$ can make a decision by selecting from the set of potentially conflicting influences, $F(i)$, the most preferred influences having the most tolerable consequences.

**Definition 2 (Decision).** Given an agent $i$, the set of influences $F(i)$ and the expected consequences $EC_i(\varphi)$ for all $\varphi \in F(i)$, we can get the set of best influences (the decision) the agent should choose from, denoted by the function $Dec : Ag \rightarrow 2^{F(i)}$ as follows:

$$Dec(i) = \{\varphi \mid \varphi \in F(i), \text{ and for all } \psi \in F(i), \psi \neq \varphi, \text{ either } \varphi \leq_P \psi, \text{ or } \varphi \approx_P \psi \text{ and } EC(\varphi) \leq_{T(\varphi \lor \psi)} EC(\psi)\}$$

Given a model $M$, an agent $i$ can then choose an arbitrary literal from $Dec(i)$, since all of these are equally preferred and with equally tolerable consequences.

If there are no expected consequences of bringing about a certain proposition, i.e. if $EC(\varphi) = \emptyset$, then we consider it tolerable, since we do not expect any consequences. Therefore for all other consequences, $\gamma$, we have to consider $\top \leq_{T(C)} \gamma$ and $\gamma \leq_{T(C)} \top$. Note that $T(\top | \varphi)$ is true iff $\varphi$ is true in any world$^3$. Furthermore, $\top \leq_P \varphi$ is always true, and $\varphi \leq_P \top$ is true iff $\varphi$ is true in all worlds. Thus it is possible to make a decision even if some obligations or desires have no known consequences.

**Lemma 1.** Given an agent $i$ and expressions $\varphi$, $\psi$, and $\gamma$, the following relation holds for relative tolerance:

$$\neg(\varphi \leq_{T(\gamma)} \psi) \Rightarrow (\psi \leq_{T(\gamma)} \varphi)$$

**Proof.** We assume $\neg(\varphi \leq_{T(\gamma)} \psi)$ and prove that $(\psi \leq_{T(\gamma)} \varphi)$. We divide the proof into two parts based on the definition of relative tolerance:

$$\neg((T(\varphi | \gamma) \land \neg T(\psi | \gamma))) \quad \text{(1)}$$
$$\neg((T(\varphi | \gamma) \leftrightarrow T(\psi | \gamma)) \land (\varphi \leq_P \psi \lor \psi \approx_P \varphi)) \quad \text{(2)}$$

1. When (1) does not hold, then we have that either $T(\varphi | \gamma) \leftrightarrow T(\psi | \gamma)$ or $\neg T(\varphi | \gamma) \land T(\psi | \gamma)$ holds. In the latter case we have that $\psi \leq_{T(\gamma)} \varphi$ by the definition of relative tolerance. Otherwise they are equally tolerable and we have to consider the second case.

$^3$ Since $T(\top | \varphi) = \mathcal{G}^P \varphi \land \mathcal{D}^P (\neg \varphi \lor \square^P (\varphi \land \top)$.
2. When (2) does not hold, then either \( \neg(T(\varphi | \gamma) \leftrightarrow T(\psi | \gamma)) \) or \( \neg(\varphi \leq_i^P \psi \lor \varphi \approx_i^P \psi) \). If the former is the case, then one is tolerated and the other is not. Since (1) does not hold, we have that \( \neg T(\varphi | \gamma) \land T(\psi | \gamma) \) and therefore \( \psi \leq_{T(\gamma)}^i \varphi \). If the latter is the case then we have that \( \neg(\varphi \leq_i^P \psi) \land \neg(\varphi \approx_i^P \psi) \).

In that case we have that \( \psi <_P \varphi \) and therefore \( \psi \leq_{T(\gamma)}^i \varphi \).

\[ \square \]

**Proposition 1.** Given an agent \( i \), a non-empty set of influences \( F(i) \) and the expected consequences \( EC_i(\varphi) \) for all \( \varphi \in F(i) \), the set of decisions is always non-empty.

**Proof.** If \( |F(i)| = 1 \) then \( |Dec(i)| = 1 \) as well, since there are no \( \psi \neq \varphi \) in \( F(i) \).

If \( F(i) \) contains more than one influence we consider two arbitrary influences \( \varphi \) and \( \psi \). We want to show that of such two influences, either one or both are chosen. If \( \varphi <_P \psi \) then \( \varphi \in Dec(i) \). If \( \varphi \approx_i^P \psi \) and \( EC(\varphi) \leq_{i}^{1} EC(\psi) \) then \( \varphi \in Dec(i) \). We proceed by showing that if neither holds, then \( \psi \) is more preferred (or its consequences are more tolerable) and it is therefore chosen.

1. If \( \neg(\varphi <_P \psi) \) then either \( \varphi \leq_i^P \psi \) or \( \psi \leq_i^P \varphi \). If both are the case, then \( \psi <_P \psi \) so \( \psi \in Dec(i) \). Otherwise we have \( \varphi \leq_P \psi \land \psi \leq_P \varphi \) or \( \varphi \not\leq_P \psi \land \psi \not\leq_P \varphi \), i.e. \( \varphi \approx_i^P \psi \) which is considered in the second case.

2. Either \( \varphi \approx_i^P \psi \) does not hold or \( EC(\varphi) \leq_{T(\varphi \lor \psi)}^i EC(\psi) \) does not hold.

In the former case we then have that either \( \varphi <_P \psi \) or \( \psi <_P \varphi \), which means that \( \varphi \in Dec(i) \) or \( \psi \in Dec(i) \) respectively. In the latter case we have that \( \neg(EC(\varphi) \leq_{T(\varphi \lor \psi)}^i EC(\psi)) \). By lemma 1 we then have \( EC(\psi) \leq_{T(\varphi \lor \psi)}^i EC(\varphi) \), and therefore \( \psi \in Dec(i) \).

Thus when deciding between two arbitrary influences, at least one will be chosen.

If the influences are equally preferred and tolerable then both will be chosen. \( \square \)

## 4 Generating models

If an agent has certain preferences, they are usually not described as a model shown above. Rather will they be expressions such as “I prefer that it does not rain” or “When it rains, I want to stay inside”. In order to utilize such preferences in the decision procedure above, a transformation is required. In the following we present a method which will generate a QDT-model that respects non-contradictory rules specified by the agent.

The model is initialized using the set of possible atoms, \( L \). We then create a model containing a world for each set in \( 2^L \), where each set either contains the atom or its negation. For instance, given \( L = \{a, b\} \), the initial model will be \( 2^L = \{\{a, b\}, \{-a, b\}, \{a, -b\}, \{-a, -b\}\} \). Certain worlds may be deemed impossible, such as raining from a clear sky. These situations are specified as prohibitions in the environment using expressions, e.g. \( \neg a \land b \). All worlds which entails such an expression are then removed from the initial set of worlds, yielding the set of possible worlds, \( W \).
Algorithm 1 Rule application

\begin{algorithm}
\begin{function}{apply}{((\varphi, \psi), W, \leq)}
\max \leftarrow \max(\leq)
\text{for all } w \in W \text{ do}
\quad \text{if } w \models \varphi \land \neg \psi \text{ then } W_c \leftarrow w
\quad \text{if } (w \models \varphi \land \psi) \text{ and } \neg \exists w' (w' \in W \land (w', w) \in lock) \text{ then}
\quad \quad o(w) = \max + 1
\quad W_s \leftarrow w
\quad \text{if } W_s = \emptyset \text{ then return } \bot
\text{for all } w \in W_s, w' \in W_c \text{ do } lock(w, w')
\text{return } \top
\end{function}
\end{algorithm}

An ordering, $\leq$, is the result of a mapping from a world to a natural number, the $o$-value, denoted $o : W \to \mathcal{N}$, such that worlds with higher numbers are more favored. Worlds can have the same $o$-value if they are equally favored. The maximum $o$-value of an ordering $\leq$ is denoted $\max(\leq)$.

Each agent specify a set of rules of the form $(\varphi, \psi)$, where $\varphi$ and $\psi$ are standard propositional formulas. A rule is to be understood as follows. Worlds $w$, in which $w \models \varphi \land \psi$, are favored over worlds $w'$, where $w' \models \varphi \land \neg \psi$. Thus a rule can be roughly interpreted as the conditionals for preference and normality. In the following we propose a method for generating preference and normality orderings which respect such rules by utilizing this interpretation. The generic definition of the conditional operators is

$$
\text{if } \varphi \text{ then } \psi \equiv \mathcal{H}\neg \varphi \lor \mathcal{G}(\varphi \land \Box (\varphi \rightarrow \psi)).
$$

From this definition it is clear that there are two ways to ensure that a rule $(\varphi, \psi)$ is respected. Either (a) $\varphi$ is never true or (b) in the most favored world(s) where $\varphi$ is true, $\psi$ is also true. Clearly (a) is easily achieved; we simply remove all worlds where $\varphi$ is true. This is however probably not what was intended by the agent, since the rules are most likely specified such that favored situations are actually also possible situations. We therefore require that the method does not remove any worlds from $W$. The method should ensure that after the application of a rule we have $M \models (\varphi, \psi)$. Another natural requirement is that previously applied rules still hold after application of a new rule. If this is not possible, we say that the new rule contradicts previously applied rule, and it is discarded.

We propose using a locking mechanism in which the ordering between two worlds can be locked, such that if $lock(w_1, w_2)$ then it must always be the case that $w_1 \leq w_2$. We can use this to e.g. lock the ordering between worlds $w_1 = \{\varphi, \psi\}$ and $w_2 = \{\varphi, \neg \psi\}$, such that if a rule $(\varphi, \psi)$ is applied, we create a lock $lock(w_1, w_2)$ such that $w_1$ is always favored over $w_2$. Then if a rule $(\varphi, \neg \psi)$ is applied, the ordering cannot be changed so that $w_2$ is favored over $w_1$ because it would result in the previously applied rule no longer being respected (since $\psi$ would not be entailed by the most favored world where $\varphi$ holds).

A rule is applied using the function $apply : (\mathcal{R}, \leq) \to \{\top, \bot\}$ (algorithm 1). Applying a rule $(\varphi, \psi)$ is done by finding all worlds in which both $\varphi$ and $\psi$
holds (the sought worlds) and all worlds in which \( \varphi \) and \( \neg \psi \) holds (the contradictory worlds). The sought worlds are given an \( o \)-value of \( \max(\leq) + 1 \) and all contradictory worlds are locked in relative position to the sought worlds.

A rule \((\varphi, \psi)\) cannot be applied if there is no world \( w \) in which \( w \models \varphi \land \psi \) or for all such worlds a lock \( \text{lock}(\_ , w) \) exists.

**Proposition 2.** Given an initial ordering \( \leq \), a set of rules \( \mathcal{R} = \{r_1, \ldots, r_n\} \) where each \( r_i \) is of the form \((\varphi_i, \psi_i)\), the result of successfully applying rules \( r_1 \) to \( r_i \), \( 0 < i \leq n \) is an ordering which respects rules \( \{r_1, \ldots, r_i\} \).

**Proof.** When \( i = 1 \) no previous rules have been applied, so we only have to show that the model respects rule \( r_1 \) after successful application. We have \( o(w) = 1 \) for all worlds \( w \). Applying \( r_1 \) can only fail if no worlds entail \( \varphi_1 \land \psi_1 \) or all entailing worlds are locked. Since \( \text{lock} = \emptyset \) initially, only the former can be the case. But then the rule would describe an impossible world and cannot be applied. Otherwise, after applying \( r_1 \), it is entailed by the model, since for all worlds \( w \) where \( w \models \varphi_1 \land \psi_1 \) we have \( o(w) = 2 \) and the \( o \)-value of all other worlds is unchanged. Thus the worlds entailing \( r_1 \) are most preferred so the rule itself is entailed by the model.

When \( i > 1 \) we assume that all rules up to and including \( r_{i-1} \) have been applied successfully. We therefore have

\[
M \models (\varphi_1, \psi_1) \land \cdots \land (\varphi_{i-1}, \psi_{i-1}).
\]

Let \((w_\varphi, w_\neg_\psi) = \{(w, w') \mid w \models \varphi \land \psi \text{ and } w' \models \varphi \land \neg \psi\}\) be the set of locks between worlds with contradictory consequences of a rule \((\varphi, \psi)\). Before applying \( r_i \) the set \( \text{lock} \) then contains

\[
\text{lock} = (w_{\varphi_1}, w_{-\psi_1}) \cup \cdots \cup (w_{\varphi_{i-1}}, w_{-\psi_{i-1}}).
\]

Rule \( r_i \) can then be applied if there is at least one world \( w \) in which \( w \models r_i \), which is not the second entry of a pair in \( \text{lock} \) (i.e. there is a world entailing \( r_i \) which is not locked by another world). If there is no such world then either the rule describes an impossible world and should be rejected, or a previously applied rule contradicts it, which also means it should be rejected. Otherwise the rule will be successfully applied resulting in a model entailing all rules up to and including \( r_i \):

\[
M \models (\varphi_1, \psi_1) \land \cdots \land (\varphi_i, \psi_i),
\]

and a new \( \text{lock} \) set: \( \text{lock}' = \text{lock} \cup (w_{\varphi_i}, w_{-\psi_i}) \). Assuming that the rule is successfully applied we know that for all \( w \) in which \( w \models r_i \) we have \( o(w) = \max(\leq) + 1 \). Clearly \( r_i \) is then entailed by the model. We then have to show that all rules up to \( r_i \) are still entailed as well.

Consider rule \( r_j \) where \( 0 < j < i \). Rule \( r_j \) was entailed by the model before applying \( r_i \). There therefore are worlds \( w_j \) where \( w_j \models \varphi_j \land \psi_j \) and no lock of it exists, and \( w'_j \) where \( w'_j \models \varphi_j \land \neg \psi_j \), and for all such worlds we have that \( o(w_j) > o(w'_j) \) and \( (w_j, w'_j) \in \text{lock} \). Thus all worlds contradicting \( r_j \) are locked relative to those entailing it. If \( w'_j \cap w_s \neq \emptyset \) then some of the sought worlds are
Algorithm 2 Model generation

function generate($L$, $P$, $R$)
    $W_0 ← \text{init}(L)$
    $W ← \text{clean}(W_0, P)$
    $≤ ← o(W)$
    $R' ← \text{sort}(R)$
    for all $(\varphi, \psi) ∈ R'$ do
        apply($(\varphi, \psi), W, ≤$)
    return $≤$

locked by $r_j$, but since $w_s$ only contains unlocked worlds, this cannot be the case. Therefore no worlds $w'_j$ will be given a higher $o$-value than any $w_j$ world. Furthermore, since $w'_j$ contains all the worlds that could invalidate $r_j$, clearly $r_j$ is still entailed after applying $r_i$.

Thus the procedure respects previously applied rules when applying new rules.

Even though a successful application of a set of rules can be done, we still need to touch upon how to maximize the number of successful applications of rules. Note that the use of a locking mechanism decreases the number of worlds that can be moved around every time a rule is successfully applied. Therefore, by minimizing the number of worlds being locked in each iteration, we maximize the number of rules that can be applied. The function $s : R → \mathcal{N}$ gives each rule a score, where rules with many propositions and operators receive higher scores than rules with few.

$$
\begin{align*}
    s((\top, \psi)) &= s(\psi) - 1 \\
    s((\varphi, \psi)) &= s(\varphi) + s(\psi) \\
    s(\varphi \land \psi) &= s(\varphi) + s(\psi) + 1 \\
    s(\varphi \lor \psi) &= s(\varphi) + s(\psi) + 1 \\
    s(\neg \varphi) &= s(\varphi) + 1 \\
    s(\top) &= 0 \\
    s(p) &= 1
\end{align*}
$$

By applying the highest valued rules (the most specialized) first, we ensure that as few worlds as possible are locked. Notice that that rules where the antecedent is $\top$ will be penalized as they are very general, whereas $\top$ in the consequent is ignored.

The algorithm generate : $(L, R) → ≤$ then works as follows (algorithm 2). Generate an initial model using init($L$). Sort rules descending according to their $s$-value using sort($R$). Each rule in $R$ is then applied using apply($(\varphi, \psi), W, ≤$). Finally, the ordering $≤$ which respects all successfully applied rules is returned.
5 Case study

We consider a situation in which agents are normally expected to go to work, but during snowy weather, they are not expected to go to work. The agent Alice prefers that it does not snow, but when it snows she wants to stay at home. Thus we have the following rules for expectations of the environment and preferences of the agent:

\[
R_{Env} = \{(\top, \text{work}), (\text{snow}, \neg \text{work})\} \\
R_{Alice} = \{(\top, \neg \text{snow}), (\text{snow}, \neg \text{work})\}.
\]

In the following we let \(S\) abbreviate \textit{snow} and \(W\) \textit{work}. We denote negation using an overline, e.g. \(\overline{S}\) when it is not snowing and conjunction is implicitly present when propositions are written next to each other, e.g. \(SW\) when it is snowing and the agent is working. From the description above it is clear that \(\mathcal{L} = \{W, S\}\). The orderings \(\leq_P\) and \(\leq_N\) are then generated using the algorithms described above. Figure 1 shows how Alice’s preference ordering is generated using her rules.

This situation is however not very interesting, since even when it is not snowing, there is no expected consequence of \textit{not} going to work. We therefore add the possibility of \textit{getting fired} \((F)\) and of \textit{leaving early} \((E)\). Alice’s rules are updated accordingly:

\[
R_{Alice} = \{(\top, \overline{S}), (S, \overline{W}), (\top, F), (W, E)\}.
\]

Thus she does not want to get fired, and in situations where she chooses to go to work, she prefers to leave early. The rules of the environment are updated to conform to this change; if it snows, one can stay home without getting fired, but this is not the case when it does not snow.

\[
R_{Env} = \{(\top, W), (S, \overline{FW}), (\overline{SW}, F), (\top, E), (W, F)\}.
\]

Note also that agents are not expected to leave early, and will normally not be fired if they work.

It should be evident that certain worlds are not possible given the new propositions; an agent will not be working if it is fired, and if it is not working, it will not leave early. We therefore express the following prohibitions to be used for cleaning the set of possible worlds.

\[
P = \{FW, E\overline{W}\}.
\]
Thus the set of possible worlds $W$ is then reduced to those worlds where none of the prohibitions above are entailed. The preference and normality orderings resulting from these rules are shown in figure 2(a) and 2(b).

5.1 Making a decision

Given the model generated above, Alice is now able to decide between her influences. Say Alice has a desire to stay at home, but an obligation towards her employer to go to work. Her influences are then $F(a) = \{W, \overline{W}\}$, where $a$ denotes Alice. Furthermore, let us consider two cases; one where it snows, and one where it does not.

a) We have that $B(a) = \{S\}$ so all worlds in which it does not snow can be ignored since Alice believes it snows. This leaves us with four possible worlds. Alice’s most preferred world in which it snows is $\overline{EFSW}$. This world is more preferred than any other world, thus $\text{Dec}(a) = \{\overline{W}\}$.

b) Alice does not believe that it snows, so we have that $B(a) = \{\overline{S}\}$. In this case Alice’s most preferred worlds are $\overline{EFSW}$ and $EFSW$. As these worlds are equally preferable, she has to consider the expected consequences of either influence. Looking at the normality ordering (figure 2(b)), we realize that $EC(W) = EFS$ and $EC(\overline{W}) = \overline{EFS}$. From the Alice’s preferences it should be clear that the expected consequences of going to work are more preferred than those of staying home (i.e. not getting fired is preferred), thus $\text{Dec}(a) = \{W\}$.

Note that at this point we have not labeled Alice as “social” or “selfish”. Her preference and the expected consequences are taking into account, and this leads to the results above. When she chooses to go to work, this does not mean that she is strictly social. She might very well have a (selfish) desire to leave early, which could be chosen if the consequences of doing so are tolerable.
6 Conclusion

We have argued that conflicts are prone to arise when agents interact in open societies and enact roles in an organization, since their own desires may be in conflict with obligations towards other agents or the obligations of the role(s) they are enacting. We have discussed why obligations along with desires should be considered influences on the agent’s behavior rather than being seen as desires being imposed onto the agent by other entities. Since obligations do not (necessarily) represent propositions the agent wants to achieve, such influences should only be pursued if their consequences can be tolerated by the agent.

Our approach to resolving such influence conflicts, which is based on qualitative decision theory, is an attempt to let the agent reason about the influences without taking into account that one influence is a desire, and another is an obligation, since such bias results in labeling the agent “selfish” or “social” in advance. This approach works by taking the consequence of bringing about a state into consideration, thus letting the agent take its preferences into account, without choosing something that results in an intolerable state. We have argued that this indeed lets the agents reach a decision without strictly preferring desires over obligations or vice versa.

To make the procedure readily available we furthermore have developed a simple method which can generate models to be used in the reasoning process by the use of expressions describing the agent’s preferences. By use of a simple locking mechanism, the method generates models which respect non-contradictory rules specified by the agent, such that it is possible to make a decision among a set of influences. The simple nature of the method also allows us to generate the models on the fly, such that if the agent’s preferences change during execution a new model can be generated. Since the method works by generating all possible states, it may prove to be inefficient in more complex cases. It would be natural to look into how this can be optimized, e.g. by considering smaller sets of propositions relevant to each rule, which would then be combined into a preference order.

A reasonable direction for further research would be to integrate the procedure into an existing agent programming language such as GOAL [7]. In GOAL the choice of committing to different goals and performing actions is quite simple; a program consists of a list of rules which are either evaluated in linear or random order. This means that either the preference ordering is specified a priori, or it is not specified at all. While requiring a specification of the rules of the agent, it should be possible to integrate the procedure in GOAL to evaluate the rules using the decision procedure such that the most preferred and most tolerable goals are pursued.

Furthermore the non-propositional case should be investigated such that more complex reasoning about the agent’s preferences can be done. For instance it should be possible for the agent to prefer being at home, \texttt{at(home)}, compared to other places such as work, while still being able to express that being at the zoo is more preferred than being at home.
References

Automatic BDI Plan Recognition from Process Execution Logs and Effect Logs

Hongyun Xu¹,³, Bastin Tony Roy Savarimuthu², Aditya Ghose³, Evan Morrison³, Qiying Cao¹, Youqun Shi¹

¹School of Information Science and Technology, Donghua University, Shanghai, 201620, P.R. China
²Department of Information Science, University of Otago, PO Box 56, Dunedin, New Zealand
³College of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW, 2522, Australia

xjyc2007@gmail.com, tonyr@infoscience.otago.ac.nz, aditya@uow.edu.au, caoqiying,yqshi@dhu.edu.cn

Abstract. Agent applications are often viewed as unduly expensive to develop and maintain in commercial contexts. Organizations often settle for less sophisticated and more traditional software in place of agent technology because of (often misplaced) fears about the development and maintenance costs of agent technology, and the often mistaken perception that traditional software offers better returns on investment. This paper aims to redress this by developing a plan recognition framework for agent program learning, where behavior logs of legacy applications (or even manually executed processes) are mined to extract a ‘draft’ version of agent code that could eventually replace these applications or processes. We develop, implement and evaluate techniques for inferring agent plans from behavior logs, with both positive and negative examples. After obtaining the plans, we resort to an effect log to identify the context (i.e. precondition) for each plan. The experimental results show that our framework generates a first draft of an agent program (i.e. the code) which can then be modified as required by a developer.

1 Introduction

Agent applications are often viewed as unduly expensive to develop and maintain in commercial contexts. Even in organizations where there is recognition that the agent technology has benefits, people often settle for less sophisticated, and more traditional software in place of agent technology because of (often misplaced) fears about the development and maintenance costs of agent technology, and the (often mistaken) perception that traditional software offers better return on investment [9]. This paper posits that agent program learning offers a solution to this problem. We define the general agent program learning problem as follows:

– Given: A behavior log describing the behavior of a system over a given audit period, and an effect log¹ describing the states of the system at some point in time.

¹ An effect log contains the resultant states of a program as it continues executing different steps. In contrast, the behavior log contains details about actions that succeeded or failed. In other
Determine: An agent program such that if that agent program were executed with the same set of inputs from the environment, the original behavior log would be obtained (with the exception of failure instances, to be discussed below, which would be avoided).

An agent program learning system can lead to significant improvements in programmer productivity. Instead of having to develop an agent program from scratch, which is an expensive and time-consuming proposition, especially in light of the well-known knowledge acquisition problem [4, 14], an agent programmer would be provided with an initial, ‘draft’ program, which the programmer could edit with far less effort relative to writing a program from scratch to obtain a complete and correct agent program. Agent technology is often deployed to ‘upgrade’ legacy applications [9]. The behavior logs required for an agent program learning system could thus be obtained by auditing the behavior of the legacy system.

A typical BDI agent program consists of three components [10]: (1) A set of beliefs, which may be dynamically generated by sensor inputs. (2) A set of plans, where each plan contains a triggering event, context conditions and planbody (a set of action sequences). (3) A set of goals that an agent wants to achieve. Each goal can be achieved by executing plans in the plan library. The goals involved are related to plans recognized and are given a unique label (e.g. goal $g1$ can be achieved by executing plan $p1$).

Thus, the general agent program learning system which is the overarching direction of our work, should have modules to support the following: (1) plan recognition; (2) goal recognition; (3) preference recognition. In this paper, we only focus on plan recognition for learning a BDI agent program, as exemplified by the BDI agent programming language AgentSpeak(L) [10]. The plan recognition activity comprises of two parts: a) planbody recognition and b) context recognition.

## 2 Background and Related Work

This section provides a brief background on Workflow nets (WF-nets) and discusses other related work.

### 2.1 WF-nets

Petri nets [8] are a graphical and mathematical modeling notation that allows users to describe business processes. Formally, a Petri net is a tuple $(P, T, F)$, where $P$ is a set of places, $T$ is a set of transitions $(P \cap T = \emptyset)$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs between the places and transitions. Let $N = (P, T, F)$ be a Petri net, $N$ is a WF-net if and only if it satisfies the following three requirements. (1) It has only one

---

2 Each agent can have a set of preferences and these preferences contribute to the selection of an appropriate plan from a set of available plans [3].

3 In other words, the plans of an agent program that are automatically created by our system will conform to Jason’s AgentSpeak specification [2].

---
input place $i \in P$, such that $\bullet i = \emptyset$.  
(2) It has only one output place $o \in P$, such that $o\bullet = \emptyset$.  
(3) If a new transition $t'$ is added to $N$, $T \cup \{t'\}$, and $t'$ is used for connecting the output place to the input place, $F \cup \{(o, t'), (t', i)\}$, such that the new net is strongly connected, $N'= (P, T \cup t', F \cup \{(o, t'), (t', i)\})$. The WF-net that we use in this work is a structured WF-net [14]. A sample WF-net is given in Figure 2.

### 2.2 Related Work

Very few work have addressed the problem of automatic recognition of plans in the area of agent-oriented software engineering (e.g. [5]). The work in [5] proposes an incremental plan recognition in an agent programming framework, which is similar to our work. However, our work is distinct from theirs. They focus on the formal model of plan recognition based on situation calculus and the ConGolog agent programming language. In their work, the plans are filtered as more actions are observed based on the existing plan library. In contrast, plans in our work are generated from scratch and it is added to an initially empty plan library set. Also, we use a process mining approach to infer BDI plans from both positive and negative examples in behavior logs produced by a traditional system. Our goal is thus to create a framework that generates plans of an agent program which when executed will produce the same behavior as that of a traditional (legacy) system. This new program is the first cut of the “agentified” version of the traditional program.

A difference of our work when compared to the work of Traverso and Pistore [13] where they convert a OWL-based business process to Hierarchical Task Network (HTN) plans is the ability of our system to derive an agent program just based on observed outputs and effects (i.e. we do not start with a process involving a particular technology, i.e. OWL-S), and our approach is generic (i.e. can involve composition). Meneguzzi and Luck [7] have investigated how context conditions can be derived for plans in AgentSpeak(PL), which is similar to our work. Even though the context conditions derived in our work are also plans in AgentSpeak(L), the process of obtaining the plan is different. Their work uses an action model for context derivation (i.e., the precondition and postcondition of actions are known), while our work requires an effect log for context derivation (i.e., the state of a system is known at different points in time). Also, the above mentioned work do not consider the identification of the erroneous conditions (i.e. the actions that caused failure) and the possibility of creating plans to handle those failures.

There exist other work where method preconditions are learned using the HTN [6, 16]. In [16], a set of constraints are constructed from observed decomposition trees under partial observations, and then solved by a constraint solver. HTN planning systems are related to BDI agent systems when it comes to know-how information used, that is, learning preconditions for a method amounts to learning context condition of a plan in BDI systems. As opposed to the work in [16], our work utilizes a complete

---

4 The expression $\bullet i = \emptyset$ denotes that there are no incoming edges to the input place $i$ (the dot here represents a set of transitions).

5 The expression $o\bullet = \emptyset$ denotes that there are no outcoming edges from the output place $o$ (the dot here represents a set of places).
behavior log and an effect log for context derivation. Also, we use propositional logic to represent the state of a system at different timestamps, and the actions are without parameters. The work in [6] learns preconditions from plan traces and HTN structures, where the task decomposition is known a priori (i.e. task dependencies are known), while in our work, we derive task composition according to the WF-net transformation rules (discussed in Section 4.2). They use a candidate elimination method to obtain a set of candidate predicates for preconditions (contextual conditions) of plans. However, in our work, we derive context conditions from effect logs obtained (i.e. resultant state of the system). More importantly, our work encompasses a higher level objective of generating a first draft of a plan library.

3 Architecture of the Plan Recognition Framework

The goal of this work is to describe how a first draft of an agent’s plans can be automatically recognized from actions recorded in the business process execution log (i.e. a behavior log) and the effect log. The process of plan recognition is shown in Figure 1. A plan consists of two main aspects, a plan body and a context (i.e. precondition that results in the execution of a plan body when it evaluates to true). These two aspects of plan recognition are carried out by two modules namely the planbody recognizer and context recognizer.

A typical behavior log contains information about both successful process executions and failures. In our work, from the log containing successful executions, we generate a workflow diagram (or WF-net6) using the process mining tool, ProM. Then, we demonstrate how a planbody recognizer (a set of transformation rules and algorithms) can be used to infer BDI plans (without context) from the WF-net in Section 4. Then, the context recognizer is used to identify the context which takes an effect log, the behavior log, and the planbody that was identified in the previous step (details provided

---

6 A WF-net is the representation of the process that could have generated the actions recorded in the behavior log.
in Section 5). The plans obtained from the positive examples are hierarchical in nature (nested plans with successful actions). Also, based on failure sequences, we demonstrate how a data mining based approach previously used for norm extraction can be used to identify sequential normative actions (obligations and prohibitions).

Thus, our contributions to the automatic plan recognition are two fold. Firstly, we show how the plans are inferred both from positive and negative examples (i.e. behavior logs containing both successful and failed actions). Secondly, we identify contexts using effect logs which form the precondition for a plan.

4 Planbody Recognition

In this section, we first describe a motivating example that is used throughout the paper. Then we present the planbody recognition, first from handling positive behavior logs containing no failure information, and then from negative behavior logs containing failure information.

4.1 Motivating Example

A typical banking system contains many business applications. We refer to two of these business applications throughout the paper. They are the processes in a banking system for loan applications and money transfers, respectively. Assume that a banking system (or an agent) receives loan applications which are handled according to the type of loan (e.g. personal and business loans). If a loan application is for a personal loan, personal information such as credit history and the risk (e.g. high, medium, low), will be evaluated and then a decision will be made. At the end of the process, the applicant will be informed of the acceptance or rejection. If an application involves a business loan, the bank will check whether the information provided is correct and also check other relevant sources in order to assess the credibility of the business entity (e.g. whether it is registered member of the chamber of commerce). The bank will also audit and evaluate the business’s assets. Then it will make a decision and also notify the business. If the banking system receives a money transfer application during the handling process for loan applications, the account of the applicant will be checked to see if the money in the account is enough to be transferred. Then it will check whether the target account specified in the application exists. Then, the money will be transferred by the banking system. In order to reference easily, we use letters A, B, C, D, E, F, G, I, K, M, N and Q to describe the following actions in the example: receive loan application, check personal loan application, check business loan application, audit assets, check credit, check risk, evaluate personal loan application, inform loan applicant, receive money transfer application, check applicant account, check target account, evaluate personal loan application.

4.2 Handling Positive Behavior Logs

In this sub-section, we describe how a behavior model is constructed from the actions available in a behavior log, and then present the transformation rules and the algorithms that transform a WF-net to BDI plans.
Behavior Model Construction - As shown in Table 1, there are five cases in a positive behavior log from a banking system where each action in the behavior log has a timestamp indicating the starting time of that action. For example, the actions associated with case 1 are \(\langle A, B, E, F, G, I \rangle\). The actions appear in the order they were executed. The actions logged for case 2 are \(\langle A, B, F, E, G, I \rangle\), and actions logged for case 3 and 5 are \(\langle A, C, D, I \rangle\), and the actions executed as a part of case 4 are \(\langle K, M, N, Q \rangle\).

<table>
<thead>
<tr>
<th>Case ID</th>
<th>Action ID</th>
<th>Timestamp</th>
<th>Case ID</th>
<th>Action ID</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>case 1</td>
<td>action A</td>
<td>(t_1)</td>
<td>case 1</td>
<td>action I</td>
<td>(t_{13})</td>
</tr>
<tr>
<td>case 1</td>
<td>action B</td>
<td>(t_2)</td>
<td>case 3</td>
<td>action D</td>
<td>(t_{14})</td>
</tr>
<tr>
<td>case 2</td>
<td>action A</td>
<td>(t_3)</td>
<td>case 5</td>
<td>action C</td>
<td>(t_{15})</td>
</tr>
<tr>
<td>case 1</td>
<td>action E</td>
<td>(t_4)</td>
<td>case 4</td>
<td>action N</td>
<td>(t_{16})</td>
</tr>
<tr>
<td>case 3</td>
<td>action A</td>
<td>(t_5)</td>
<td>case 2</td>
<td>action F</td>
<td>(t_{17})</td>
</tr>
<tr>
<td>case 2</td>
<td>action B</td>
<td>(t_6)</td>
<td>case 5</td>
<td>action D</td>
<td>(t_{18})</td>
</tr>
<tr>
<td>case 4</td>
<td>action K</td>
<td>(t_7)</td>
<td>case 2</td>
<td>action E</td>
<td>(t_{19})</td>
</tr>
<tr>
<td>case 1</td>
<td>action F</td>
<td>(t_8)</td>
<td>case 3</td>
<td>action I</td>
<td>(t_{20})</td>
</tr>
<tr>
<td>case 5</td>
<td>action A</td>
<td>(t_9)</td>
<td>case 4</td>
<td>action Q</td>
<td>(t_{21})</td>
</tr>
<tr>
<td>case 1</td>
<td>action G</td>
<td>(t_{10})</td>
<td>case 2</td>
<td>action G</td>
<td>(t_{22})</td>
</tr>
<tr>
<td>case 3</td>
<td>action C</td>
<td>(t_{11})</td>
<td>case 5</td>
<td>action I</td>
<td>(t_{23})</td>
</tr>
<tr>
<td>case 4</td>
<td>action M</td>
<td>(t_{12})</td>
<td>case 2</td>
<td>action I</td>
<td>(t_{24})</td>
</tr>
</tbody>
</table>

The plan recognition framework uses the ProM tool to construct a behavior model, i.e., a WF-net that describes the behavior of a system based on the actions recorded in the behavior log. ProM enables the extraction of information from event logs, and it supports several process mining techniques in the form of plug-ins, such as the Alpha-algorithm [14, 15]. We exploit the Alpha-algorithm in ProM to automatically generate a WF-net from the log.

The WF-net shown in Figure 2, is constructed from the log given in Table 1 using the Alpha-algorithm in ProM. The start and end transitions are added automatically when there is no common start action (or actions) and no common end action (or actions) among cases in the behavior logs\(^7\). The other transitions (rectangles) in the diagram are the actions that an agent must perform. The places (circles) in the WF-net are labeled \(p_1, p_2, \ldots, p_{14}\) for easier referencing.

Transformation Rules and Algorithms - WF-nets are commonly used to represent a process which is composed of various applications (e.g., different business applications in the context of the banking example). The business process when executed will produce different output depending upon the input given to the process (e.g., the path taken for evaluating a personal loan will be different from the path taken for evaluating

\(^7\) This is required to tie-together distinct set of processes a system can execute (e.g. a business system can run ten different processes at any point of time) and a WF-net is a representation of all these different processes.
money transfer application). That is, the business process will produce different result (i.e. output) depending on the triggering event (i.e., input). We argue that the behavior exhibited by the traditional system could be viewed as the behavior exhibited by an agent that follows some plans (i.e., plans for different goals are different). Different triggering events could be handled by distinct plans. In Figure 2, the condition in place $p_2$ could be triggered by different external events such as submission of personal loan application and money transfer application. Hence, we argue that, it is reasonable to view the number of branches emanating $p_2$ as the number of plans for different goals. That is, the paths that originate from $p_2$ can be different, where each path represents a different goal (e.g., the plan involving actions KMNQ represents the goal of transferring money from one account to another). In our work, each inferred plan is a BDI plan, and it consists of a sequence of actions and/or subgoals. Before presenting the algorithms for generating plans for a goal, we first present the types of sub WF-nets and the related rules for inferring plans from WF-nets.

A sub WF-net starts at a node $N$ and ends at node $N'$ where $N, N' \in P \cup T$ (as per usual convention $\bullet N = \emptyset$ and $N'\bullet = \emptyset$). The number of branches deviating at $N$ is greater than 1, i.e. $|N \bullet| > 1$. At $N'$ all of the branches join together, i.e., $|\bullet N'| > 1$. A sub WF-net is called WNOP (WF-net for Optional Plans) iff $N, N' \in P$, highlighted using solid boxes in Figure 2. And, a sub WF-net is called WNPP (WF-net for Parallel Plans) iff $N, N' \in T$, highlighted using the dotted box in Figure 2. The transformation rules for inferring plans from WF-nets are as follows.

1. The number of top-level plans for achieving different goals are determined by the number of branches emanating the node (the place), that is located next to the start transition.
2. Each transition is viewed as an action in a plan, in other words, a transition node has the same label as an action.
3. Each sub WF-net is considered as a subgoal in a plan.
4. If a subgoal is of WNOP type, the number of possible plans achieving the subgoal is determined by the number of branches emanating the start node in the sub WF-net, and these obtained plans are alternative plans, of which only one will be chosen for execution at run-time.
5. If a subgoal is WNPP type, the number of possible plans relies on the number of branches emanating the start node in the sub WF-net, and these plans can be executed in parallel.
Algorithm 1 Recognize plans for one goal

Input: wf, i.e., a WF-net; currentNode
Output: planList, i.e., BDI agent-oriented plans

1: while currentNode ≠ wf.endNode do
2:   if currentNode.type is transition AND currentNode.type ≠ wf.startNode then
3:     put currentNode.name in a plan
4:   end if
5:   if count(currentNode.outgoingEdges) = 1 then
6:     currentNode ← currentNode.nextNode
7:   else if count(currentNode.outgoingEdges) > 1 then
8:     generate a new subgoal in a plan
9:     if currentNode.type is place then
10:        planList ← obtain plans for WNOP goal
11:     else if currentNode.type is transition then
12:        planList ← obtain plans for WNPP goal
13:     end if
14:   end if
15: end while
16: planList ← planList U {plan}
17: return planList

Algorithm 1, presents the process of inferring plans for one goal. The WF-net is encoded as a graph, and the node after start transition in the WF-net is viewed as currentNode. Each node has its type, i.e., place and transition, and its name. If the node type is transition, its name will be viewed as an action label, which will be added in a plan, following rule (2). If there exists a sub WF-net, a new subgoal is generated in the plan, following rule (3). If a subgoal is WNOP type, the number of plans for the subgoal depends on the number of branches emanating the start node of the sub WF-net, as described in rule (4). Each branch constructs a new plan for the subgoal, recursively using the Algorithm 1. Likewise, for the WNPP type of subgoal, the number of plans for this subgoal depends on the number of branches emanating from start node of the sub WF-net, as presented in rule (5). There is no parallel construct in Jason [2], hence we handle the WNPP type of subgoal using the interleaved actions among the paths to obtain all possible plans. In the running example, two actions E and F can be executed in parallel. It means that, there are two possible ways in which these actions could have been executed, E followed by F or F followed by E.

The results of using Algorithm 1 when the WF-net shown in Figure 2 is given as the input (recursively for each goal) are given in Table 2. There are six plans in total (p0 to p5). We can observe that there are two top-level goals achieved by these plans, i.e., goal1 and goal2 (rule 1). The first goal, goal1 can be achieved directly using plan p0, while the second goal goal2 can be achieved using p1 that has a subgoal subgoal1.

---

8 If there are multiple goals, the algorithm will be used iteratively (which is the case in our work for the WF-net given in Figure 1). Also, note that top-level goals (first iteration goals) are represented as goal1 and goal2. The next level goals are represented as subgoal1, subgoal2 and so on.
There are two alternative ways to achieve this subgoal (either using plan \( p_2 \) or using plan \( p_3 \)). Note that plans \( p_0 \) and \( p_1 \) are of WNOP type. So are plans \( p_2 \) and \( p_3 \) that are used to achieve \( \text{subgoal1} \). On the other hand plans \( p_4 \) and \( p_5 \) are of WNPP type. They are used to achieve \( \text{subgoal2} \). Note that the preconditions for each of the plans is set to true. Identifying preconditions of these plans is discussed in Section 5.

<table>
<thead>
<tr>
<th>Table 2: Results of Planbody Recognition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(@p_5 + !\text{subgoal2} : \text{true} \leftarrow e; f.)</td>
</tr>
<tr>
<td>(@p_4 + !\text{subgoal2} : \text{true} \leftarrow f; e.)</td>
</tr>
<tr>
<td>(@p_3 + !\text{subgoal1} : \text{true} \leftarrow b; !\text{subgoal2}; g.)</td>
</tr>
</tbody>
</table>

### 4.3 Handling Negative Behavior Logs

In this sub-section, we discuss how plans are extracted from a behavior log that contains failure information. Failures could be caused when obligated actions do not occur or when prohibited actions are performed\(^9\). Obligated and prohibited actions can be identified using ONI and PNI algorithms. The two algorithms (ONI and PNI) that employ data-mining techniques \([1]\) have been previously used to identify obligation and prohibition norms respectively. In our work, we use them to infer plans from negative examples (failed actions in a behavior log).

The starting point for ONI and PNI is sanction recognition. Once sanctions are recognized, the reasons for these sanctions are investigated (i.e. norm violations are reasons for sanctions to occur)\(^10\). These algorithms were originally proposed to be used in artificial agent societies where interacting avatars can automatically learn norms based on observation of actions. For example, an agent that litters a park might be sanctioned by another agent. An observer, based on actions observed can infer that littering is prohibited. Also, an agent that does not tip (a violation of an obligation) might be sanctioned. PNI and ONI algorithms identify two norms, i.e., prohibit(litter) and obliged(tip)) respectively.

Failures in business process executions are logged in behavior logs (e.g. failure of a task due to a mechanical error or human error). These failures are similar\(^11\) to sanctions. When these failures happen, the reasons for the failures can be investigated. In our work we assume failures happen either because a prohibited action is performed

---

\(^9\) We acknowledge that there could be other reasons for failures such as a printer failing because of power failure. We do not model those because those type of failures are explicit failures and the reasons are known to the agent (i.e., power failure is the reason for failed printing job).

\(^10\) PNI algorithm \([12]\) identifies those actions that occur 100% of the time before the sanctions. ONI algorithm \([11]\) identifies those actions that fail to occur whenever a sanction occurs. In order to identify those actions it compares two lists of action sequences, one containing missing actions (followed by sanctions) and the other containing the expected actions (without sanctions). On comparing these lists, the algorithm identifies missing actions.

\(^11\) When a sanction happens, a special event ($) gets recorded in the norm identification framework. Similarly, in this work, when a failure happens a special event (⊗) will be recorded.
(i.e. the action that does not fit in a sequence, such as sending rejection notification before decision is made) or an obliged action does not happen (e.g. credit check is not performed before a decision can be made). We use slightly modified versions of PNI and ONI to identify the reasons for failures (prohibited and obliged actions respectively). The modifications made are two fold. First, we eliminate the norm verification stage (where an agent asks another agent to verify whether a norm holds) because we only consider an action to be prohibited by setting the threshold for norm identification to be 100% (i.e. only those actions that have the probability of 1 to be causing violations are considered). Second, we create two lists to handle the two types of norms separately (details discussed in Algorithm 2).

PNI and ONI algorithms not only identify norms but also preconditions and postconditions of norms. This is useful in our context because if we view agents as entities being norm aware (e.g. action $x$ is prohibited and action $y$ is obliged), then they need to make plans to adhere to norms. Hence, the precondition and postcondition can be viewed as parts of a plan for achieving a goal (normative goal in this case). We refer the readers to the details of PNI and ONI algorithm in [11, 12].

Algorithm 2, describes the process for identifying plan sequences from a behavior log with failure using these two norm learning algorithms. First, the correct entries without failure information are stored in a correct list, and are also removed from the behavior log (lines 4-9). Second, the entries with failure information are classified into two lists, prohibition list and obligation list. If a failed sequence (i.e. a case) has a matching super sequence(s) in the correct list, then that sequence will be added to the obligation list\(^{12}\). Otherwise, it will be added to a prohibition list (lines 10-17). Third, the ONI and PNI algorithms are invoked (lines 18, 19). Given the correct list and the obligation list as inputs to the ONI algorithm, it produces plan sequences as outputs. Next, the PNI algorithm is invoked. Given the correct list and the prohibition list as inputs, it produces plan sequences as output.

To demonstrate how Algorithm 2 works, a sample log with failure information is given in Table 3. The algorithm first creates three empty lists (lines 1-3), the correct list, prohibition list and the obligation list. The correct list is then populated with four entries: ABEFGI, ABFEGI, ACDI, KMNQ\(^{13}\). Then the algorithm populates the obligation list which contains the following entries: ABEFI\(\otimes\), BFEI\(\otimes\), ACI\(\otimes\), ABEG\(\otimes\). The prohibition list contains the following entries: KZ\(\otimes\), KMNY\(\otimes\), KMK\(\otimes\), ACDL\(\otimes\). Then, the ONI and PNI are executed in sequence to identify plans. The results of using the norm learning mechanisms are shown in the last column of Table 3.

There are two benefits of the outputs obtained from the norm learning mechanisms. First, the outputs are the correct sequences (i.e. corrected failure sequences). For example, ABEFI\(\otimes\) has been corrected to ABEFGI. So, the first benefit of norm learning mechanisms is their ability to correct erroneous sequences. Second, the output pro-

\(^{12}\) Assume actions $a,b$ and $c$ were supposed to happen in sequence. Let us assume that the observed sequence is $ac$, resulting in a failure. Here action $b$ is the obligated action. By identifying the supersequence of $ac$ (which is $abc$), we are able to identify the obliged action (i.e. $b$).

\(^{13}\) Note that $\otimes$ is used to indicate failures in the behavior log. If there is a failure, then other steps of the process are not executed.
Algorithm 2 Identifying plans from negative behavior logs

**Input:** bl, i.e., a behavior log containing failures
**Output:** ps, i.e., plan sequences with obligated actions or prohibited actions

1: Let correctList ← ∅ ▷ entries without ⊗
2: Let prohibitionList ← ∅ ▷ entries with prohibited actions
3: Let obligationList ← ∅ ▷ entries with missing actions
4: for each entry E ∈ bl do
5: if E contains no ⊗ then
6: Add E to correctList
7: Remove E from bl
8: end if
9: end for
10: for each entry E ∈ bl do
11: remove ⊗ in E
12: if E has super sequence(s) in correctList then
13: Add E to obligationList
14: else
15: Add E to prohibitionList
16: end if
17: end for
18: ps ← ONI Algorithm(correctList, obligationList)
19: ps ← PNI Algorithm(correctList, prohibitionList)
20: return ps

Produced by the algorithms can be viewed as a plan which contains a precondition, an obligated/prohibited action (or actions) and a postcondition. For example, the failure corrected version of ABEFI⊗ is ABEF-Obliged(G)-I where the hyphen is used as a separator between the three parts of a plan.

In the future, agents in our framework can use the results generated to create plans that ensure those already generated plans do not violate the normative action(s) (e.g. none of the existing plans should execute action L after executing actions A, C and D (entry 12 in Table 3)). This can be particularly valuable when the human programmer modifies the first draft of agent code generated by our framework to suit application needs (i.e. have plans to capture errors accidentally introduced by the humans that could potentially violate the norms). We note that some results obtained are incomplete (e.g. ABEG⊗ is corrected to be ABEGF and ABFEG). The complete sequence must be ABEGFI and ABFEGI. However, we address this problem by identifying supersequences of the results in the correct list to produce the correct and complete sequence.

A prohibition sequence only contains a precondition and a normative action because there could be many different post conditions depending upon which action should have occurred in the place of the prohibited action.
Table 3: A Log with Failure Information

<table>
<thead>
<tr>
<th>Entry ID</th>
<th>Sequence of Actions</th>
<th>Results from Alg. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABEFGI</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ABFEGI</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ACDI</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ABEFI ⊗</td>
<td>ABEF-Obliged(G)-I</td>
</tr>
<tr>
<td>5</td>
<td>ABFEI ⊗</td>
<td>ABFE-Obliged(G)-I</td>
</tr>
<tr>
<td>6</td>
<td>ACI ⊗</td>
<td>AC-Obliged(D)-I</td>
</tr>
<tr>
<td>7</td>
<td>ABEG ⊗</td>
<td>AB-Obliged(F)-EG ABE-Obliged(F)-G</td>
</tr>
<tr>
<td>8</td>
<td>KMNQ</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>KZ ⊗</td>
<td>K-prohibited(Z)</td>
</tr>
<tr>
<td>10</td>
<td>KMNY ⊗</td>
<td>KMN-prohibited(Y)</td>
</tr>
<tr>
<td>11</td>
<td>KMK ⊗</td>
<td>KM-prohibited(K)</td>
</tr>
<tr>
<td>12</td>
<td>ACDL ⊗</td>
<td>ACD-prohibited(L)</td>
</tr>
</tbody>
</table>

Table 4: Sample effect log

<table>
<thead>
<tr>
<th>Time</th>
<th>States</th>
<th>Time</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>c3 ∧ c4</td>
<td>t13</td>
<td>c11</td>
</tr>
<tr>
<td>t2</td>
<td>c5 ∧ c6</td>
<td>t14</td>
<td>c13 ∧ c14</td>
</tr>
<tr>
<td>t3</td>
<td>c3 ∧ c4</td>
<td>t15</td>
<td>c12 ∧ ¬c20</td>
</tr>
<tr>
<td>t4</td>
<td>c1 ∧ c7</td>
<td>t16</td>
<td>c17</td>
</tr>
<tr>
<td>t5</td>
<td>c3 ∧ ¬c4</td>
<td>t17</td>
<td>c1 ∧ c7</td>
</tr>
<tr>
<td>t6</td>
<td>c5 ∧ c6</td>
<td>t18</td>
<td>c13 ∧ c14</td>
</tr>
<tr>
<td>t7</td>
<td>c15</td>
<td>t19</td>
<td>c7 ∧ c8</td>
</tr>
<tr>
<td>t8</td>
<td>c7 ∧ c8</td>
<td>t20</td>
<td>c11</td>
</tr>
<tr>
<td>t9</td>
<td>c3 ∧ ¬c4</td>
<td>t21</td>
<td>c18</td>
</tr>
<tr>
<td>t10</td>
<td>c9 ∧ c10</td>
<td>t22</td>
<td>c9 ∧ c10</td>
</tr>
<tr>
<td>t11</td>
<td>c12 ∧ c20</td>
<td>t23</td>
<td>c11</td>
</tr>
<tr>
<td>t12</td>
<td>c16</td>
<td>t24</td>
<td>c11</td>
</tr>
</tbody>
</table>

5 Context Recognition

In Section 4, we described how plans are recognized. However, the preconditions of all the plans were true. Recognizing preconditions is a key part of plan recognition. We note that preconditions cannot be inferred from the behavior log alone. So, we assume, in addition to behavior log we also have the effect log. Behavior log contains actions that were executed and effect log contains the state of the system. For example, the state of the system can contain information such as the loan application of customer 1 was in the pending allocation state at time $t_1$ and the state was changed to assigned to risk analyst at $t_2$. By using effect log in conjunction with the behavior log and the plan body of all the recognized plans, we demonstrate how preconditions for the plans can be identified.

We assume that the sample effect log given in Table 4 can be obtained during the execution of a traditional system. We model states using propositional logic (states represented as a conjunction of propositions). We also assume that timestamps of these states are recorded. For example, at timestamp $t_1$, $c_3$ and $c_4$ hold (assume $c_3$ is application received and $c_4$ is data verified for completeness).

Algorithm 3 presents how preconditions of plans are recognized. The input to the algorithm are a) the behavior log, b) the effect log and c) planbody of all the recognized plans. The algorithm contains three main steps.

Step 1 (lines 4-8) - For each plan, if it has subgoals, all the possible action sequences for subgoals are identified. This is done by unfolding subgoals recursively to the inner most plan with subgoals. The identified set of these action sequences will be stored in an array called planTrace. Each entry in the planTrace is a possible execution sequence of a plan (i.e. actions that would be executed if a plan were to be invoked). Note that there could be more than one plan trace for a plan because a subgoal of a plan produce different results. If a plan has no subgoals, the action sequence itself will be stored in planTrace as a plan trace. For example, let us consider plan $p_5$ which does not have any subgoal. In this case, the planTrace contains actions ef. However, for plan $p_3$
Algorithm 3 Mining preconditions in plans

Input: a) sblog - a behavior log; b) elog - an effect log; c) plans resulting from planbody recognition

Output: plans with context

1: for each plan ∈ plans do
2:    planTraces[ ] ← ∅
3:    stateArr[ ] ← ∅
4:    if plan.planbody has subgoal then
5:        planTraces ← obtain all possible plan traces
6:    else
7:        planTraces ← plan.planbody
8:    end if
9:    for each trace ∈ planTraces do
10:       for each case ∈ sblog do
11:          if case contains the trace then
12:              \( T_a \) ← obtain the timeStamp of first action of the plan
13:          if elog contains \( T_a \) then
14:              stateArr ← states at \( T_a \) should be adding to stateArr
15:          end if
16:       end if
17:    end for
18:    end for
19:    if stateArr.size > 1 then
20:        plan.context←compute common proposition
21:    else
22:        plan.context←stateArr
23:    end if
24: end for
25: return plans

that contains a subgoal in its plan body, there will be two plan traces (befg and bfeg) since subgoal2 can be realized in two different ways.

Step 2 (lines 9-18) - For each plan trace in planTrace, if there exists a case in the behavior log, that contains either the same sequence of the plan trace, or the supersequence of the plan trace, the timestamp of the first action in plan trace is stored in \( T_a \) (a variable), and the entry corresponding to \( T_a \) in the effect log is stored in an array called stateArr. Note that there could more than one one result.

For plan \( p_5 \), the planTrace contains the action sequence \( ef \) as described above. There exists only case (case 1) containing the supersequence of \( ef \) which is abefgi. Then we consider the first action of \( ef \), i.e., action \( e \), and obtain its timestamp in case 1, which is \( t_4 \). So, the context for \( p_5 \), is the entry corresponding to \( t_4 \) in the effect log, which is \( c_1 \& c_7 \) (which gets stored in stateArr). For plan \( p_3 \), there are two plan traces befg and bfeg as described above. For the plan trace befg, case 1 has its supersequence, abefg. We obtain the timestamp corresponding to the first action \( b \) which is \( t_2 \). The entry corresponding to \( t_2 \) in the effect log is \( c_5 \& c_6 \) which is stored in stateArr. Similarly, case 2 contains the supersequence of \( bfeg \) which is abfeg. The timestamp corresponding to
the first action $b$ is identified which is $t_6$. The entry $c_5 \& c_6$ in effect log at $t_6$ is obtained. Note that for plan $p_3$, there are two entries in $stateArr$ and both these entries have the same propositions ($c_5$ and $c_6$).

**Step 3** (lines 19-23) - For a given plan, if the $stateArr$ has only one entry, then the entry is the precondition of the plan. If there are more than one entry, then the propositions that are common to these entries will be computed to be the context. For example, for plan $p_3$, there were two entries in the $stateArr$. Common propositions among these two entries are chosen as the preconditions for $p_3$ which are $c_5$ & $c_6$ in this case. For plan $p_5$, there is only one entry in the $stateArr$ ($c_1 \& c_7$) which becomes its precondition.

### 6 Experiments and Results

Using the simplified motivating example throughout the paper, the plans resulting from context recognition are shown in Table 5, applying the positive behavior log given in Table 1 and the effect log given in Table 4.

We can see that there are two different top-level goals to achieve for an agent, under different contexts (i.e. $goal_1$ and $goal_2$). The context (i.e. precondition) is the same for the parallel plans, $p_4$ and $p_5$, but is different for the optional plans, $p_2$ and $p_3$. However, only one of parallel plans should be executed at run-time. Since there is no parallel construct in AgentSpeak(L), in our implementation, one of these plans is randomly picked for execution. Note that the plans $p_2$ and $p_3$ for achieving $subgoal_1$ have different context conditions which mean that only one of them will be executed at run time.

#### Table 5: Results of Planbody and Context Recognition

<table>
<thead>
<tr>
<th></th>
<th>@ $p_5$ +!$subgoal_2$ ::c1&amp;c7 ← e; f</th>
<th>@ $p_4$ +!$subgoal_2$ :c1&amp;c7 ← f; e</th>
<th>@ $p_3$ +!$subgoal_1$ :c6&amp;c5 ← b;$subgoal_2$;g</th>
<th>@ $p_0$ +!$goal_1$ :c15 ← k;m;n;q</th>
<th>@ $p_2$ +!$subgoal_1$ :c12 ← c;d</th>
<th>@ $p_1$ +!$goal_2$ :c3 ← a;$subgoal_1$i</th>
</tr>
</thead>
</table>

In order to evaluate the plans generated by our plan recognition framework, we employed human subjects. A pilot study was conducted with eight participants. A small-sized problem specification (including the functional requirements such as the high level goals of the agent program and the expected behavior in terms of output) was provided to the participants, who had to handwrite the BDI agent program to achieve these goals\textsuperscript{15}. We divided the participants into two groups of four programmers. For one group (group A), we provided just the specification. For the second group (group B) we provided both the specification and the resulting ‘draft’ agent program code generated by our plan recognition framework. Our results show that the average time for programmers in group B to finish the program is much shorter than that of programmers in group A. On average, programmers in group B finished 12 minutes earlier than group A. The maximum time taken to finish the program in group A was 36 minutes, but in

\textsuperscript{15} The participants were students pursuing their postgraduate research work in computer science.
group B it was only 17 minutes. Furthermore, the average number of errors in group B was less than that of programmers in group A. The errors made by programmers in group A include the wrong ordering of actions and assigning wrong preconditions, which can be easily avoided by making use of the plans inferred by our framework. Thus, the initial results obtained from the pilot study is promising. However, extensive studies with complex requirements with large number of participants are required to firmly establish our initial findings.

7 Conclusions and Future Work

In this paper, we have proposed a novel plan recognition mechanism where BDI-style plans are generated by our plan recognition framework. This plan recognition is a part of a larger scope project which aims to learn agent programs (i.e. automatically generate a first draft of an agent program) from the behavior log and the effect log produced by a traditional ('legacy') system. The two main aspects of our plan recognition framework are the planbody recognition and the context recognition. In order to generate plans, first, a WF-net is generated from a behavior log using ProM. Second, we have proposed a set of transformation rules and a procedure (Algorithm 1), which transforms the WF-net into a set of plans (without context). Then, we demonstrated how the preconditions for the plans can be identified using an effect log (Algorithm 3). It should be noted that we have demonstrated how both positive and negative examples can be used to obtain plans. We leveraged existing norm learning mechanisms [11, 12] to infer normative plans (with prohibited and obligated actions) in the context of handling negative examples (Algorithm 2).

We have demonstrated that our plan recognition framework creates a ‘draft’ agent program which can then be extended by a programmer. We have conducted a pilot study with eight participants and the results of the study are encouraging. We believe the work presented here is an important step for BDI-type agent systems development since it shows that agent programs can be developed for existing traditional systems (or at least a draft version of the system can be developed). Especially, these agent programs can be considered as a viable alternative for ‘legacy’ systems that need to be redeveloped in an appropriate language.

The plan recognition framework has some simplifying assumptions. First, business processes with loops have not been considered in this work. That forms the focus of our future work. Second, the propositional logic is used to demonstrate the feasibility of the system. Other more expressive logics could be investigated in the future. Also, parameterized actions and states can be included in the behavior log and effect log respectively. In the future, we plan to evaluate our plan recognition framework with complex applications (complex WF-nets). We believe it is in those complex systems, our framework will offer significant advantages (i.e. reduction of programming time and effort). We will also conduct extensive testing involving substantial number of developers.

References

Agents for the 21st Century: the Blueprint Agent Programming Language*

Alex Muscar
University of Craiova, Blvd. Decebal, nr. 107, RO-200440, Craiova, Romania

Abstract. While there are plenty of agent languages around, those who are targeted towards practical aspects are few and far between. Most alternatives serve mainly as experimentation vehicles for Distributed Artificial Intelligence (i.e. Jason, GOAL). It is our opinion that if agents are to get any mainstream attention they have to come with solutions to current problems like concurrency and distribution such as those found in real world scenarios, e.g. cloud and mobile systems. While some of the existing solutions partially address some of these issues, they do so in an unconvincing way.

In this paper we propose BLUEPRINT, an agent oriented programming language with built-in support for concurrency. Other key features are static typing and checked communication protocols which prevent a part of the errors that can appear in agents developed with existing languages. We walk through BLUEPRINT’s main features, shortcomings, and future extensions, motivating our design decisions.

1 Introduction

Agent-Oriented Programming Languages (AOPLs) appeared in the context of Distributed Artificial Intelligence (DAI), and they served mostly as experimentation vehicles for ideas in this area. As such, most of the effort in developing the languages has gone into software models such as Belief-Desire-Intention (BDI), while more pragmatic aspects were superficially explored at best (e.g. typing, concurrency). It is our opinion that if the agent paradigm is to gain traction with a broader audience, AOPLs need to cater to the above mentioned aspects as well. Our view is supported by recent efforts such as [1], and [2]. The former work explores the use of the Erlang virtual machine as a platform for BDI agents, while the latter proposes a new agent language called SimpAL which addresses some shortcomings of existing agent languages such as weak type systems. The need for better agent languages is also clear from experience reports such as [3], which describes the experience of using Jason [4], one of the best known agent languages, for developing a medium software project, and the problems faced by the author. For an in-depth review of agent languages we refer the reader to [5].

* This work was supported by the strategic grant POSDRU/CPP107/DM11.5/S/78421, Project ID 78421 (2010), co-financed by the European Social Fund Investing in People, within the Sectoral Operational Programme Human Resources Development 2007-2013.

1 http://www.erlang.org/
2 http://jason.sourceforge.net/
In this paper we propose Blueprint\textsuperscript{3}, an agent oriented programming language with built-in support for concurrency. Blueprint mostly resembles Jason (the adoption of concepts such as beliefs, plans and goals) but it also draws its inspiration from JADE (the reactive aspects of agents and some of the message passing details). Blueprint could be considered a dialect of Jason, but it has some important differences which we will discuss in more detail in sec. 3 and sec. 4. As such we expect the user to be at least familiar to Jason’s syntax in order to understand the code listings in this paper. We refer those who are not familiar to Jason to the excellent resource [4].

We do not present a formal model of Blueprint in this paper\textsuperscript{4}, but one can get a rough idea of our model by reading [6], which serves as a major inspiration for our work. Another relevant work is [7], which presents a semantics for the concurrency primitives we have adopted for beliefs in Blueprint.

Our motivation in conceiving Blueprint comes from our experience using Jason to develop a generic negotiation framework [8], as well as the experience of teaching agent development using the JADE framework\textsuperscript{5} to undergraduate students. The experience with Jason has guided our language design decisions, while JADE’s shortcomings influenced some lower-level, runtime related aspects. While at the different ends of the spectrum—Jason being a BDI AOPL, and JADE a reactive agent framework—we find that the synergy between the above mentioned sources of inspiration has had an overall beneficial role in the design of Blueprint. Another source of inspiration was the Sing# programming language [9]. Some of the ideas embodied by Blueprint have been presented in the context of the Jason programming language in an earlier work [10].

Blueprint has the following design goals (we will focus more closely on them in sec. 2 and sec. 3):

**Concurrency** Since there seems to be a lot of confusion in this area we would first like to define the term “concurrency” as it is used in this paper: *concurrency is the interaction of independently executing entities.* This definition covers a lot of scenarios, from small mobile applications, to large cloud systems. Furthermore, concurrency is intrinsically present in a Multi-Agent System (MAS), both inside the MAS (i.e. multiple agents executing concurrently), and inside the agent (e.g. multiple behaviours executing concurrently as is the case in JADE). As such, we decided to make handling concurrency as natural as possible in Blueprint;

**Scalability** When we talk about “scalability” we are actually referring to two different aspects: (i) the ability of the runtime to gracefully handle a growing number of agents executing concurrently; and (ii) the ability of the language to gracefully handle a growing project;

**Safety** This point is closely related to the previous one, more exactly item (ii): having the knowledge that some invariants are maintained (e.g. knowing that the message flow in an agent system remains valid when extending it, knowing that the received message has the desired content) enables developers to tackle larger projects. Some of these aspects can be handled by static type systems, a feature which has only

\textsuperscript{3} As of writing this paper the Blueprint compiler is still under development, and the source code can be found at the following address \url{https://github.com/muscar/blueprint}.

\textsuperscript{4} Though such a model is under development.

\textsuperscript{5} jade.tilab.com
recently been seriously considered in the AOPL community (e.g. SimpAL has a static type system [2], AF-Raf is an agent language with algebraic data types [11]);

**Expressivity** We tried to keep close to the agent-oriented metaphor in designing Blueprint so we have adopted concepts such as beliefs, plans and plans. Furthermore we have tried to make situations frequently encountered in agent applications (e.g. complex communication scenarios) easily expressible by the developer; and

**Interoperability with the host platform** Developing a new programming language comes with some heavy tolls, one of which is the lack of libraries. One way around this is for the language to target an already existing platform (e.g. the Java Virtual Machine (JVM) or .NET), and benefit from the already existing libraries. This is the route chosen by Jason, which runs on the JVM. But there is a tension between Jason’s high level semantics and JVM’s lower-level features so developers using Jason frequently need to use an “escape hatch”: they need to drop into Java to develop parts of their agent system. This is a dangerous route since the developer might be tempted to use the path of least resistance, and develop the better part of the agent system in Java (this issue is also covered in [3]). One of the goals of Blueprint is to make the native libraries easy to access from the language itself, and the language expressive enough to make their use comfortable.

The rest of the paper is organised as follows: in sec. 2 we take a closer look at our driving examples and their shortcomings, Jason and JADE. Section 3 introduces the Blueprint programming language in more detail through the use of some source code samples. Some of the design decisions are discussed more in-depth in sec. 4. We finally conclude in sec. 5 with some closing remarks and future research and development directions.

## 2 Background and Related Work

In this section we take a closer look at our driving examples, Jason and JADE, and their shortcomings. We are going to structure the rest of this section around the design goals of Blueprint (see sec. 1). Before we begin though, we will shortly introduce the computation models of both Jason and JADE.

Jason uses the concept of beliefs, which represent the agent’s view of itself and the world, goals, which represent states in which the agent wants to be, and plans, which are “recipes” for action. An agent is driven forward by a reasoning cycle in which the agent: (i) perceives the environment, and updates its beliefs, (ii) receives messages, (iii) selects an event and the plans that can handle it, and (iv) executes (a step of) the plan. Note that running plans are organised as intentions, which are stacks of currently executing plans. The Jason interpreter uses a single OS thread for executing all the intentions, thus a single intention is active at any given time. While this is a (very) simplified description of the reasoning cycle, it has all the information that we need.

Message passing in Jason is quite complex, messages featuring illocutionary forces which give the semantics of the communication act. Message sends are asynchronous, except for when the ask illocutionary force is used, in which case the current plan is suspended until a reply is received or a timeout occurs. Message receives are also
asynchronous, i.e. there is no explicit receive operation, instead Jason handles message receive by generating an internal event which can be handled by a plan written by the programmer (with the obvious exception of using the ask illocutionary force).

JADE uses the concept of behaviour to drive agent execution. An agent can have multiple behaviours which are cooperatively scheduled, i.e. a behaviour will execute until it decides to suspend itself. Both synchronous and asynchronous message receive is available. As a peculiarity of JADE’s design, if the non-blocking version of the receive operation is used there is the risk that the current behaviour occupies the thread indefinitely, thus blocking other behaviours from executing. As such, developers are encouraged to use a pattern in which an asynchronous read is immediately followed by a check to see if a message was actually received, and blocking the behaviour if that is not the case [12]. Each agent has its own OS thread on which it executes all its behaviours in a Round Robin fashion.

2.1 Concurrency

The main unit of execution in Jason is the agent. Agents are ran concurrently, all multiplexed on the same OS thread. An agent’s plans are most commonly ran sequentially, but the language provides a mechanism to run a plan asynchronously, by prefixing it with the !! operator. The only way to synchronise plans is by using the belief base. This approach is sub-optimal for a couple of reasons: (i) it mixes declarative knowledge representation with operational aspects, i.e. condition variables, and (ii) it only works because the agents are all serviced by the same OS thread. The latter point is important because it affects the interoperability of Jason code with Java code that updates the belief base of the agent from other threads. Care must be taken by the programmer to avoid such situations or the risk of corrupting the belief base appears. For a more in-depth discussion of these issues we refer the user to [10].

Unlike Jason, the main unit of execution in JADE is the behaviour. An agent’s behaviours are multiplexed on the same OS thread. This is a step in the right direction, as an agent solution developed using JADE can take advantage of the full concurrency available on the host system. However, the decision to use one OS thread per agent proves to be a limiting one: because OS threads are expensive there is only a relatively low number that can be spawn before the system starts to under-perform. The main synchronisation point for JADE agents is the message receive operation. But, as mentioned before, JADE offers a blocking version of message receive which makes synchronisation possible without using workarounds like shared state, as is the case in Jason. JADE agents are exposed to the same risks as Jason agents when faced with OS threads that are not part of their runtime: a thread spawned by the user, or by library code, can corrupt the state of the agent. For a more in-depth discussion of these issues we refer the reader to [13].

2.2 Scalability

As we stated previously, the agent is the main unit of execution in Jason, and they are ran concurrently. This leads to a certain pattern of formulating solutions to problems, where agents are used as a non-blocking mechanism of doing a part of work. Synchronisation
is then performed via the agents’ belief base (for an in-depth treatment of this situation see [10]).

While it is possible to synchronise concurrently running agents in Jason by using their belief bases, it is not optimal for a couple of reasons:

1. It splits the logic of the program over several execution units which makes it hard to have a global view of the program behaviour, especially for more complex scenarios;
2. Dedicating one agent to each task that needs to be accomplished leads to inefficient and hard to manage programs. While several strategies are possible – starting all the task-executing agents up-front, dynamically creating and destroying them, or managing a pool of such agents – none of them is optimal, the first two because of the cost of running extra agents, and the third because of the extra complexity involved in managing a pool of client agents;
3. It does not scale: having to manually store intermediary results as beliefs for many asynchronous requests and synchronising them by hand is a tedious and error prone task and it leads to mostly duplicate code for handling asynchronous responses; and
4. Having to structure large programs with callbacks, i.e. Jason’s message handlers, leads to the problem known as “inversion of control”, a situation where the caller is not in control of its own program flow, but instead the callee is.

The previous points apply to the ability of the language to handle a large code base that implements concurrency heavy scenarios, as is the case of most applications developed with agents. As mentioned in the previous section, the main bottleneck of runtime scalability in Jason is the use of a single OS thread to service all executing agents.

JADE’s ability to handle large programs is pretty much that of Java, so there are no issues in this area, but the use of one OS thread per agent limits the number of agents that can run on the same platform and machine, and can prove to be a bottleneck in some scenarios.

2.3 Safety

Being a latently typed programming language, Jason is prone to all the risks of the languages that adopt this typing discipline. While simple errors such as not passing the right number of arguments to a call, or passing an argument of a wrong type can be spotted quite easily, there are subtler cases, like message handlers not getting called because of an arity mismatch, or a unification failure because of incompatible types. These sort of problems slow down development, and can lead to hard to diagnose and repair bugs in programs.

Another problem faced by both Jason and JADE is the lack of static verification of message flow. Indeed, this is one of the main pain-points for our undergraduate students when developing JADE applications for performing various kinds of negotiations. In some scenarios, because of faulty program logic the wrong message will get sent, resulting in an out-of-sequence exception being thrown, with little information pointing to the real cause of the problem.
2.4 Expressivity

There are two approaches to enriching the set of concepts that the programmer can use while developing an agent program: (i) developing a custom language which features all the intended concepts as first class; and (ii) developing the intended semantics in the form of a framework. Jason is probably the best known AOPL. As mentioned before it is implemented and extensible in Java. While it scores high on expressivity, it more or less lacks when it comes to generality. The best argument for this is that the only way to extend it is using their implementation language.

In the case of JADE, which uses Java, a generic programming language, to implement agent programs it is hard to express concepts that are specific to the metaphor (e.g. agents, acts of speech) in a concise way. The best a developer can hope for is to factor out recurring patterns into libraries. Even though JADE offers “primitives” for MAS development (e.g. agents, interaction protocols) the conceptual and syntactic noise level is quite high. In order to implement an agent the developer has to extend a base class and override some methods. These concepts belong to the implementation language, not to the agent language. They are implementation details that the developer should not have to cope with, they are signs of leaky abstractions. The need for a higher level of abstraction arises.

2.5 Interoperability with the Host Platform

While JADE applications can make use of all the libraries available in Java directly, in Jason the developer has to write his or her own wrappers around the functionality needed, and expose them as actions. This approach has two main disadvantages: first, it makes the developer use two languages instead of one, and second, it exposes implementation details specific to the Jason interpreter which the agent program developer should have no reason to dwell into. Writing a Jason internal action feels more like extending the interpreter itself than writing a component of the agent program.

3 The BluePrint Programming Language

BluePrint has been designed in order to address some of the issues discussed in sec. 1, and sec. 2, specifically concurrency, and static safety. We will review the most important aspects of BluePrint in the following subsections, and motivate our design decisions.

3.1 Concurrency and beliefs

Listing 1 shows the implementation of the well known bank account example in BluePrint. Although simple, this problem is interesting because it exposes some of the pitfalls of concurrent programming, namely race conditions (incidentally this is also an example of a purely reactive agent).

We will use the following scenario to illustrate the problem: the initial balance of a bank account is $100, agent A sends a withdraw message for $40, while another agent, B, sends another withdraw message for $80. The agent managing the bank account...
starts handling the first withdraw message, reads the value of the balance, and \(b\) gets bound to the value 100. Now, the thread executing the handler for the first withdraw message gets interrupted (e.g. by the OS scheduler), and the thread handling the second withdraw message starts running, and it also read the balance value, the value bound to its own \(b\) variable is also 100. It goes on to update the balance to 20, and finishes. After some time, the thread handling the first withdraw message wakes up, and continues where it left off and updates the balance to 60—remember that the value it holds in \(b\) is 100. Clearly this is not the desired behaviour. This is a classic example of a race condition.

agent Account(Init: int, ImpChan: BankAccount.Imp) {
  balance(Init).

  +!start
  <- .make_chan(BankAccount, ExpChan, ImpChan)
    .recv(ExpChan, Msg);
    (case Msg of
     | deposit(Amount) ->
       ?balance(B); +balance(B + Amount)
     | withdraw(Amount) ->
       ?balance(B); +balance(B - Amount)
     | transferTo(Acc, Amount) ->
       ?balance(B);
       .send(Acc, deposit(Amount));
       +balance(B)).
}

Listing 1: A bank account agent.

There are various solutions to race conditions such as the one illustrate previously. The most common approaches are: (i) shared-state concurrency, and (ii) message passing. The former usually involves explicitly managing the locking of the shared state in order to avoid possible race conditions, while the latter uses a higher level approach, that eschews explicitly managing locks by not exposing the shared state, and serialising message execution. While message passing is clearly more in tune with the high level approach advocated by AOP we find that using message passing for expressing the internal logic of agents comes with a high price on programmer productivity since internal state manipulation happens relatively frequently. Given these reasons we chose a third, in-between, approach: we represent the internal state of the agent as synchronised mutable variables as found in Concurrent Haskell\(^6\) [7], which in turn are based on M-Structures as presented in [14]. In keeping with the nomenclature introduced by Concurrent Haskell we will call these mutable variables mvars.

The semantics of mvars is relatively simple. They are one-place buffers which can be in one of the two states: empty or full, and support two basic operations: take, and put.

\(^6\)http://www.haskell.org/haskellwiki/GHC/Concurrency
Calling \textit{take} on a \textit{full} mvar immediately returns the value and marks the mvar as \textit{empty}. If a \textit{take} call is issued on an \textit{empty} mvar, the calling thread of execution is blocked until the mvar becomes \textit{full}. The semantics of the \textit{put} operations are similar: performing a \textit{put} on a \textit{full} mvar blocks until the mvar is empty, while performing the \textit{put} on an \textit{empty} mvar immediately succeeds, and marks the mvar as \textit{full}, allowing other threads of control waiting on the mvar to resume their execution. While mvras exhibit a blocking behaviour, the locks are not directly manipulated by the programmer, instead this is the job of the underlying implementation.

\textsc{Blueprint} implements all beliefs as mvras, so in our previous example \texttt{balance} is an mvar. In keeping with the Jason notation, the two operators used to manipulate beliefs: \texttt{?} (read as “query”), and \texttt{+} (read as “update”) correspond to calls to \textit{take}, respectively \textit{put}. Given the previously description of the operational behaviour of mvras, the implementation of the \texttt{Account} agent in listing 1 is free of race conditions. Unfortunately, the same can not be said about deadlock freedom. Given the relatively low level, blocking nature of mvras (when compared to message passing), the risk of deadlock is still present. We need not look further than \texttt{transferTo} to see the risks. We will use the following scenario to illustrate a situation in which a deadlock can occur: given two agents \texttt{Acc1}, and \texttt{Acc2}, managing two accounts, a third agent, \texttt{A}, can initiate a transfer from the account managed by \texttt{Acc1} to the account managed by \texttt{Acc2}, and another one in the opposite direction. Given the “right” order of execution of the operations in \texttt{transferTo}, the two agents \texttt{Acc1}, and \texttt{Acc2} can end up infinitely waiting for each other to release the \texttt{balance} mvar. This serves to illustrate the important point that “there is no such thing as a free lunch”, the relative ease with which race conditions can be avoided comes at the price that deadlocks can still occur.

Before we go any further a note is due: our proposed solution is an improvement with respect to concurrency, but it does not solve the fundamental issue of logical belief consistency. While this is an issue that deserves a lot of attention it is not the purpose of this research to address it. We are predominantly interested in the concurrent aspects of agent programs.

### 3.2 Message Passing and Channels

While we find channels to be too heavy weight for intra-agent operations, we find the paradigm to be the natural choice expressing patterns of interaction in a MAS. Unlike process calculi such as CSP \cite{15}, and the $\pi$-calculus \cite{16}, \textsc{Blueprint} uses \textit{asynchronous} channels, i.e. the sender does not wait for the acknowledgement that the message was received. This design decision is motivated by: (i) the fact that is bears a closer resemblance with the \textit{actor model} \cite{17}, to which the agent model bears undeniable resemblances, and (ii) asynchronous channels are less prone to deadlock. The latter point is more interesting. In order to see how it is motivated we can look at it from the perspective taken in \cite{18}: first, we do not talk about procedures, or operations, we talk about plans\textsuperscript{7}; second, during a plan’s execution, if it needs to cooperate with another plan in order to achieve its final goal, it has either the option to immediately interrupt its execution, and wait for the other plan do its part of the work, or it can postpone contacting

\textsuperscript{7} This is in tune with the nomenclature used by the agent community.
the other plan until a later time, when its execution is complete. The *immediate* strategy is more prone to deadlocks, because the interrupted plan’s progress depends on the other’s plan progress, while the *postpone* strategy allows for the plan to make progress until it is absolutely necessary to wait for the other plan, thus minimising the risk of deadlock. This strategy is not bullet proof though since it may as well lead to deadlocks in pathological scenarios (see sec. 3.1).

Channels in Blueprint are *bidirectional* and *asymmetric*. A channel has two *endpoints*: an *exporting* endpoint, and an *importing* endpoint. This design decision has been influenced by the design of channels in Sing# and Axum [9,19], and it is necessary in order to allow static checking of message flows (see sec. 3.3 about protocols). The exporting endpoint is used by the *owner* of the channel, while the importing endpoint is used by other agents wanting to exchange message with the owner of the channel. Blueprint uses the .send primitive to send messages, which takes a channel and a message to be sent on the channel. Messages are received in the order they were delivered. The .recv primitive is used to synchronously receive messages, i.e. in a blocking fashion. The first argument must be a channel, while the second can be either a compound term, in which case it is unified with the received, or a variable. Our example in listing 1 uses the latter, and the case construct to pattern match on the contents of the message.

The names of the messages must correspond to those specified by the protocol adopted by the channel (see sec. 3.3). The fact that communication only occurs on already defined names, as well as the asynchronous nature of message passing brings Blueprint closer to such process calculi as the Join-calculus [20].

### 3.3 Protocols

Protocols are declarative mechanisms of enforcing proper message exchange between agents with respect to some desired semantics. Their design is influenced by *channel contracts* as found in the Sing# and Axum languages [9,19] and by *session types* [21,22]. Protocols specify: (i) the name of the messages that are used in the interaction, (ii) the type of data that is sent with the messages, and (iii) the flow of the data between the communicating entities, i.e. the *order*, and *direction* in which messages are sent. The concept is best illustrated by an example (see listing 2).

```plaintext
protocol EnglishAuction {
  start: !cfp(int) ->
    ?propose(int) ->
    (!accept_proposal or !reject_proposal)
    >> [start or end];
}
```

**Listing 2**: A protocol for simplified English auctions.

Listing 2 shows a simplified protocol for an *English auction*, a *one-to-many* protocol, where a distinguished agent called the *initiator* interacts with many agents called *participants*. 
Protocols are introduced by the `protocol` keyword. A protocol contains one or more `message flow expressions` with an associated `label`. The label `start` is mandatory for all protocols, and it corresponds the starting point of the message exchange process. Message flow expressions obey the EBNF grammar given in listing 1.

flow_exp ::= {exp}[^>] » [' state_exp ']
state_exp ::= {label}[^or]
  exp ::= label | message | choice | '(' exp ')'
choice ::= {exp}[^or]
message ::= direction ident ['(' args ')']
  args ::= {type}[^{]}
direction ::= '!' | '?'
label ::= ident

Figure 1: Message flow expressions grammar.

A message flow expression is obtained by enumerating the messages that are supposed to be exchanged in the desired order—ordering is expressed by `[^>]`. In order to pass the static checks messages must be exchanged in the specified order. Next come the `direction specifiers`, `!'` for `send`, and `'?'` for `receive`. Even though this notation overlaps the one for belief query we have kept to it since it seems to be standard in the field of checked protocols and session types. We believe the risk of confusion is relatively low since the notation is used in syntactically separate contexts. A message expression prefixed by a `!'` means that the agent owning the channel `adopting` the protocol must send a message with that specific name on the channel, while a message prefixed by `'?'` signifies that a message with the specified name must be received on the channel. Messages can carry data, which is expressed in the protocol by the type of data which can be exchanged via a channel. This is the only important information since protocols are tied in the type system—the number and types of arguments to a message are statically checked (see sec. 3.4).

Protocols, as described so far are not expressive enough to represent a series of desirable flows. Looking at English auctions (our running example for protocols) we see that there are two more features that we need: first, we need to able to `choose` between multiples messages at some points of the interaction, and, second, we need to express `looping`. The former is obtained by using the `choice operator` (or) which states that either of its operands is valid at that point of the interaction—the choice operator is left associative. Note that the parenthesis around the choice operators in the sample listing are used in order to avoid ambiguity. The latter problem is solved by the use of `states`. States can be used to abstract message expressions by using a name in order to aid their re-use, and in order to allow recursive definitions of message flow expressions, thus enabling looping. Every state definition in a protocol must end with a state transition, introduced by the `[^]` operator. This is illustrated in listing 2 by the state transition `[^start or end]`. When the protocol has reached this point there are two acceptable

---

8 We extend the standard EBNF notation with the following construct: `{p}[^sep]`, to stand for a sequence of zero or more productions `p` separated by the terminal `sep`. 
actions: either transition to the start state (i.e. the current state, effectively achieving looping), or transition to the end state, which is the built-in state that can be used to signal the end of message flows. Even though our example only uses one state a protocol can use any number of states necessary to make it as readable as desired.

Protocols are designed from the perspective of the agent initiating the interaction, i.e. the exporting endpoint. There is no need to specify the dual protocol, i.e. from the perspective of the importing agent, since it can be automatically derived by swapping direction specifiers. Channels are said to adopt protocols. When a channel is created using the .make_chan primitive, the desired protocol is passed to the primitive, which creates the channel, and returns its two endpoints. The exporting endpoint can be used by the agent creating the channel, while the importing endpoint can serve as a capability to communicate with this agent. The agent can pass it to any other agent with which it wants to communicate, i.e. channels are first order entities in Blueprint. This can be used to implement a model similar to object capabilities [23].

Listing 3: Implementation sketch for English auction initiator and participant.

```plaintext
agent Initiator(Ah: AuctionHost.Imp) {
    quote(0).
    +!add_participants(0, _).
    +!add_participants(Cnt: int, [P|Ps])
    <- .recv(Ah, participate);
    .make_chan(EnglishAuction, P, ImpChan);
    .send(Ah, ImpChan);
    +!add_participants(Cnt - 1, Ps).
    +!start_auction([]).
    +!start_auction([P|Ps])
    <- +!negotiate(P);
    +!start_auction(Ps).
    +!negotiate(P: EnglishAuction.Exp)
    <- ??quote(Q);
    .send(P, cfp(Q));
    .recv(P, propose(Offer));
    ?quote(R);
    +!check_offer(P, R, Offer, NewQuote);
    +negotiate(P).
    +!check_offer(P: EnglishAuction.Exp,
        CurrentQuote: int,
        Offer: int, Offer)
    : CurrentQuote < Offer
    <- .send(P, accept_proposal).
    +!check_offer(P: EnglishAuction.Exp,
        CurrentQuote: int,
        Offer: int, CurrentQuote)
    <- .send(P, reject_proposal).
    +!start
    <- !add_participants(3, Ps);
    +!start_auction(Ps).
}

agent Participant(Ah: AuctionHost.Imp) {
    amount(1000).
    +start
    <- .send(Ah, participate);
    .recv(Ah, initiator(Initiator));
    +negotiate(Initiator).
    +negotiate(Initiator: EnglishAuction.Imp)
    <- .recv(Initiator, cfp(Price));
    ?amount(Amount);
    +check_cfp(Initiator, Price, Amount, NewAmount);
    +amount(NewAmount);
    .recv(Initiator, Response);
    +!negotiate(Initiator).
    +check_cfp(Initiator: EnglishAuction.Imp,
        Price: int, Amount: int,
        NewAmount: int)
    : Price < Amount + 10
    <- .send(Initiator, propose(Amount + 10));
    NewAmount = Amount - 10.
    +check_cfp(Initiator: EnglishAuction.Imp,
        _, Amount: int, Amount)
    <- .send(Initiator, refuse).
}
```

Listing 3: Implementation sketch for English auction initiator and participant.
Listing 3 shows the implementation of simple English auction initiator and participant agents. Note that the code implements the model of English auctions adopted in [8,24] in which a special component, the auction host acts as an intermediary between initiators and participants.

There are some interesting aspects of Blueprint that show up in the code so we will discuss them in turn.

- The initiator agent is handed the capability to communicate with the auction host, an agent managing ongoing auctions. The type of this capability is AuctionHost.Imp, which assuming that there is a AuctionHost protocol defined, denotes its importing endpoint. The .Exp qualifier denotes the exporting endpoint of a channel;
- Blueprint uses the !! operator to asynchronously execute a plan, just like Jason (the scheduling details differ, we will come back to this matter in sec. 4);
- Blueprint introduces the ?? operator to read the value of an mvar. The read operation suspends the calling thread if the mvar is empty, and immediately returns the value of the mvar otherwise, without emptying it;
- The use of beliefs as synchronisation mechanisms can be seen if the negotiate plans of both agents. As an example, take the Participant agent: the amount belief is emptied, the agent checks if it can further bid, then it fills the belief with the new amount if it decides to bid further, or with the old amount otherwise.

3.4 Static Typing

Blueprint is a statically, strongly typed programming language with a form of parametric polymorphism [25] similar to the generics system in C# or Java, and limited type inference [26]. Listing 3 illustrates all the features mentioned above. Specifying the type of variable parameters for agents and plans is mandatory. The reasoning behind our decision is that we want to keep the type inference algorithm simple, and the types also act as documentation. The types of beliefs and local variables are inferred from their initialisation sites. As an example, the quote belief in the Initiator agent is inferred to have type int.

Parametric polymorphism is the most interesting feature of Blueprint’s type system. This allows to make agent code generic and reusable, while still maintaining type safety. The second argument of the add_participants plan has type List[EnglishAuction.Exp]. List is a generic container which can be parametrized with the type of its elements. While this is also possible in latently typed agent languages, Blueprint also guarantees that the operations performed on the contents of the container are in accordance with the type of the elements in contains. The expression .send(P, cfp(Q)) will type check, and it is guaranteed to not throw a runtime exception, because the compiler knows that P is of type EnglishAuction.Exp, thus it can handle cfp messages.

4 Discussion

Blueprint borrows features from both Jason, and JADE. In fact it could be considered a dialect of Jason, but it has some important differences. The language retains the
notions of plans, beliefs, goals, and intentions, but it does not include Jason’s rule
language, nor strong negation. While useful in writing plan contexts, the rule language is
orthogonal to plans and goals, thus not essential [4, p. 38-39]. Other significant departures
from Jason are the changed semantics of .send, and the introduction of an explicit,
synchronous .receive. We feel that these departures make concurrent programming in BLUEPRINT easier, even if they come at the price of giving up the illocutionary force
system and their rich semantics. Some of the functionality offered by Jason’s .send can
be regained by using specific interaction patterns in BLUEPRINT.

The two most important aspects of BLUEPRINT are its built-in support for concurrency
that integrates quite well with the existing Jason model, and its added support for static
safety. In our design we have tried to keep close to the Jason syntax and idioms, making
only the necessary extensions and additions. Such examples are the agent construct
used for encapsulation, and the case construct for easier pattern matching inside a plan.

4.1 Implementation considerations

We are currently in the process of implementing a compiler for BLUEPRINT on the .NET
platform. We have chosen this platform because of its rich set of libraries, its portability,
and our previous experience with developing such projects on the platform. We also plan
to have a version running on the JVM, but for the moment our efforts are targeted toward
the .NET implementation. In order to support the style of programming exemplified in
listing 3, i.e. sequential programming, with blocking message receives without blocking
the agent we will employ a scheme similar to that presented in [6]. In short, we will
perform a rewrite of the source code in Continuation Passing Style (CPS) [6], and
use a thread-pool [27] for scheduling reactions to pending events, e.g. communication.
This model has proven quite effective; it has been employed by the F# programming
language since 2007, and recently C# has adopted a similar solution. Instead of forcing
the developer to write the code in explicit CPS, which is what Jason does by requiring
separate event handlers for each message received, we will let the compiler take care of
the plumbing.

5 Conclusions and Future Work

In this paper we have proposed a new AOPL, similar enough to Jason to be considered
dialect, but with important differences, mainly the addition of strong support for
concurrency and static safety. The latter is obtained by adding a static type system, and
channel protocols in order to check the order in which messages are sent and that their
payload type checks.

There are a couple of aspects which we haven not touched on in this paper, but which
are on our immediate research agenda. First, we want to investigate modularity and
encapsulation in the context of BLUEPRINT. While JADE benefits from Java’s strong object
oriented properties, Jason is lacking with respect to this matter. We want to investigate
what the best approach is for BLUEPRINT. Second, we want to investigate code reuse for
our language, a topic which is mostly neglected by the agent community. Third, we plan
to look into other methods of sharing state between agents in the same address space.
besides the use of channels. A recent effort in this direction is presented in [28]. Last, but not least, we plan to give a formal account for our language in the future. In this regard we find [24] very interesting in the context of protocol verification.

References

Belief Caching in 2APL

Mehdi Dastani and Marc van Zee

Utrecht University
Department of Information and Computing Science
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands

Abstract. The BDI-oriented multi-agent programming language 2APL allows the implementation of an agent’s beliefs in terms of logical facts and rules. An agent’s beliefs represent information about the agent’s surrounding environment including other agents. Repeated querying of the beliefs by the 2APL interpreter causes unnecessary overhead resulting in poor run-time performance of the interpreter. We propose a heuristic to reduce the number of such queries by using belief caching. We show that our belief caching implements and extends an existing caching proposal. Benchmarking results indicate that belief caching can lead to significant improvements.

1 Introduction

The multi-agent programming language 2APL\(^1\) supports the implementation of individual agents that can perform high-level reasoning and deliberation about their information (i.e., beliefs) and objectives (i.e., goals to achieve) in order to decide what actions to perform [3]. Beliefs and goals in 2APL are declarative; Beliefs are represented by a set of Horn clauses and goals are represented by conjunctions of first-order atoms. While this allows the development of flexible and declarative agent programs, repeated inferencing triggered by queries to the beliefs can result in poor performance. When developing multi-agent systems for time critical applications, performance issues are often a key concern, potentially adversely impacting the adoption of BDI-based agent programming languages and platforms as an implementation technology [1]. For example, if agent programming languages want to provide better support for implementing autonomous robots, one of the requirements is real-time reactivity to events, which is currently lacking [6].

We present an inference method based on caching within the 2APL interpreter that reduces the number of belief queries. Our motivation for this approach is based on the observation that belief queries are responsible for most of the deliberation time within a 2APL deliberation cycle and that most belief queries are redundant because they are being performed repeatedly while relevant parts of the belief base do not change, meaning the result of such queries

\(^1\) For more information, see: http://apapl.sourceforge.net/.
will be the same. Using the notion of caching is therefore likely to be an optimization. Recently, it is shown [1] that it is theoretically possible to improve the run-time execution of BDI-based agent programs by using belief caching. However, this proposal focuses purely on the optimization of belief queries within one so-called update cycle, which consists of a query phase and an update phase. Also, the proposal has not been implemented. Our approach specializes the concept of a general update cycle to an update cycle for each individual query that may cover multiple deliberation cycles. We show that the general update cycle is contained in our proposal and that our proposal is more fine-grained leading to an increased number of queries answered by the cached beliefs.

We implement belief caching in the 2APL interpreter by performing a belief query only if the belief base has been updated in a way that is relevant to this query. We exploit the fact that both belief queries and belief updates are static in 2APL, which makes it possible to determine what belief update will change what belief query at compile-time. In order to do this, we define the notion of relevance for belief queries by making use of query dependency sets in the belief base. We have implemented our belief caching approach into the latest version of 2APL. Additionally, we have implemented a generative benchmarking tool, which allows the reader to test the working of belief caching very easily. The manual for the benchmarking tool can be found in the 2APL manual.

The structure of this paper is as follows. In Section 2 we introduce 2APL together with the parts that are relevant to our analysis. In Section 3, we introduce our belief caching approach, compare it with the abstract performance model as proposed in [1], and show how our approach can be seen as an extension to this work. Finally, we provide implementation details and benchmarking results in Section 4.

2 2APL - A Practical Agent Programming Language

The programming language 2APL is developed to implement multi-agent systems [3]. In 2APL, individual agents are programmed in terms of beliefs, goals, actions, plans, events, and three types of practical reasoning rules. The beliefs and goals of 2APL agents are implemented in a declarative way, while plans are implemented in an imperative style. The declarative part of the programming language supports the implementation of an agent’s reasoning task and the update of its mental state. The imperative part of the programming language facilitates the implementation of plans, control flow, and mechanisms such as procedure call, recursion, and interfacing with legacy codes. 2APL agents can perform different types of actions such as belief and goal update actions, belief and goal test actions (belief and goal queries), external actions (including sense actions) and communication actions. The practical reasoning rules can be applied to generate plans. The first type of rules is designed to generate plans for achieving goals (so-called Planning Goal rules, or PG rules), the second to

\footnote{The sources of the latest 2APL version can be downloaded from \url{http://www.apapl.sourceforge.net/}.}
process external events, messages and abstract actions (so-called Procedure Call rules, or PC rules), and the third to process internal events for repairing failed plans (so-called Plan Repair rules, or PR rules). Each practical reasoning rule has a belief query that specifies the belief state in which the rule can be applied.

2APL agents are autonomous in the sense that they continuously deliberate on their mental states (beliefs, goals and plans) in order to decide which plans to select and execute. This deliberation mechanism, which is an integral part of the 2APL interpreter, iterates over a reasoning cycle, depicted in Figure 1. The reasoning cycle starts by applying applicable PG rules of an agent program in order to generate plans to achieve the agent’s goals. The reasoning cycle continues by executing the generated plans. Then, the received internal and external events and messages are processed by applying PC and PR rules. We would like to emphasize that the application of all practical reasoning rules as well as the execution of belief test actions require queries to the belief base. The fact that the application of practical reasoning rules is the core activity of each reasoning cycle implies that the belief query actions constitute the most frequent operations in the reasoning cycle. Therefore, any significant reduction in the number of belief queries is expected to improve the performance of the 2APL interpreter.

![Fig. 1: The 2APL deliberation cycle](image)

**2.1 Belief queries**

Belief queries can occur at two places in a 2APL program: as guards in the practical reasoning rules and as belief query actions in a plan. We will discuss both of them separately. In what follows, we denote a belief query with $\beta$ and substitutions with $\tau$. 
Practical reasoning rules. As mentioned, 2APL programs may involve three kinds of practical reasoning rules each of which contains a belief query. The three types of practical reasoning rules share the same syntax. A practical reasoning rule in 2APL has the form $H \leftarrow \beta \mid \pi$ where $H$ is the head of the rule, $\beta$ is the guard of the rule representing a belief query, and $\pi$ is the body of the rule representing a plan. The representation of $H$ is different for each rule type. In case of the PG rule, $H$ is a goal expression represented by a conjunction of positive first-order atoms. For a PC rule, $H$ is either a message, an event or an abstract action represented by a first-order atom. Finally, in case of a PR rule, $H$ is a plan whose execution has failed and is represented by a sequence of actions containing variables. The belief query $\beta$ may contain conjunctions and disjunctions of first-order literals. A successful query of this guard results in a substitution that can be applied to instantiate variables that occur in the body of the rule. Finally, $\pi$ is the plan that will be added to the plan base if the rule is applied. The complete description of 2APL constructs can be found in [3].

An example of a 2APL program is depicted in Figure 2. This program consists of a single agent that will repeatedly move towards and away from a target until it runs out of fuel. The distance $X$ of the agent from the target is represented by the belief fact $\text{dist}(X)$. Initially, the agent is halfway from the target (the distance is 50) and will start moving forward with a speed of 5 (represented by the
goal `driveForward(5)`). This will select the first PG rule, which is applied with
the substitution `[Speed/5]` resulting from the unification of the head with the
goal base, and the substitution `[D/50]` resulting from the unification of the belief
query in the guard of the rule with the belief base. This rule is repeatedly applied
until the agent reaches the target (`D <= 0`). Then the goal `driveForward(5)`
will be replaced with the goal `driveBackward(NewSpeed)`, where `NewSpeed` is a
random integer between 0 and 10. This will activate the second PG rule, which
does exactly the opposite as the first PG rule. This process will repeat until the
agent runs out of fuel (`enough_fuel(Speed)` can no longer be entailed from the
belief base).

As this example might suggest, practical reasoning rules are applied in 2APL
in the following way. First the head is instantiated, resulting in a substitution,
which we will denote by `τ1`. In case of our example, when applying the first
PG rule with the head `driveForward(Speed)`, this results in substitution
`τ1 = [Speed/5]`. Subsequently, the substitution `τ1` is applied to the guard of the rule
creating a new belief query which in case of our example is `enough_fuel(5)` and
distance(`D`). Note that the application of `τ1` to the guard of a rule does not
necessarily instantiate all variables involved in the guard (in case of our example
variable `D`) such that querying the guard to which `τ1` is applied can result in a new
substitution, which we will denote by `τ2`. Finally, we would like to emphasize that
if there are multiple substitutions for a query possible, then the first substitution
is returned. In the case of our example, the new substitution is `τ2 = [D/50].`

Belief Test Action A belief test action occurs in a plan and checks whether the
agent has certain beliefs. A belief test action is an expression of the form `B(φ),`
where `φ` is a belief query represented by a conjunction or disjunction of first-
order literals. The execution of a belief test action is basically a belief query to
the belief base that can generate a substitution. Since a belief test action occurs
in a plan, it may be preceded by some other actions that share variables. This
means that some of the variables of a belief test action may already have value
instantiation through earlier computed substitutions, which we denote by `τ1`
(e.g., substitution resulted from the guard of the practical reasoning rule whose
application has generated the plan, or from earlier belief test actions in the same
plan). Similar to practical reasoning rules, we first apply the earlier computed
substitution `τ1` to the query of the belief test action and then use the new query
to check the belief base. The new query will result in a new substitution which
we denote by `τ2`.

In the case of our example, the belief test action `B(D <= 0 and new_speed
(NewSpeed))` contains the variable `D` that is instantiated when the PG rule is
applied. This means that `τ1` will contain a substitution for `D`. It also contains the
variable `NewSpeed` that is not instantiated before the belief query is performed,
which means that it will be instantiated by the belief query. Therefore, `τ2` will
contain a substitution for `NewSpeed`. 
2.2 Belief updates

2APL contains two different types of belief update actions. The first type of belief update action requires a belief update specification. Each belief update specification is characterized by a triple consisting of the action name represented as an first-order atom starting with a capitalized letter, a precondition represented by a set of first-order literals, and a post-condition that is also represented by a set of first-order literals. One of the belief updates of the example in Figure 2 is:

\{ \text{dist}(X) \text{ and } \text{fuel}(F) \} \text{ Forward}(Y) \{ \text{not dist}(X), \text{not fuel}(F), \text{dist}(X - Y), \text{fuel}(F - Y) \}

This triple specifies that any belief update action that unifies with this action name (e.g. \text{Forward}(5)) can be executed when the pre-condition can be derived from the belief base (when \text{dist}(X) \text{ and } \text{fuel}(F) can be derived from the belief base for some substitution of the variables \(X\) and \(F\), for instance \text{distance}(50) and \text{fuel}(1000)). The execution of the belief update action ensures that the post-condition is derivable from the belief base (e.g. \text{not dist}(50) and \text{not fuel}(1000) and \text{dist}(45) and \text{fuel}(995) is derivable from the belief base after the execution of \text{Forward}(5)). Note that the action call \text{Forward}(5) will instantiate the variable \(Y\) and that variable \(Y\) in the post-condition is instantiated with the same value.

The second type of belief update action does not require a belief update specification and consists of a first-order atom preceded by either the plus (+) or the minus (−) operator. An update action with the plus operator adds the atom to the agent’s belief base while an update action with the minus operator will remove the atom from the agent’s belief base. For example, the plan "−\text{dist}(50); +\text{dist}(45);" will remove the fact \text{dist}(50) from the belief base and add the fact \text{dist}(45) to it. Note that the syntax of simple update actions is the same as the syntax of belief updates in Jason [2].

3 Extending 2APL with Belief Caching

In the previous section, we explained that belief queries demand a substantial amount of processing time of each deliberation cycle and we analyzed belief queries and belief updates in 2APL in order to infer when the result of a belief query will not change and caching can be applied. The answer of a belief query remains unchanged if the following three conditions are satisfied: 1) the part of the belief base that is relevant for the query is not changed, 2) in the case of a practical reasoning rule where the head and the guard share variables, the unification of the head provides a substitution that assigns the same values as the cached values to the shared variables, and 3) in the case of a belief test action that shares variables with some actions that precede it, the substitution originating from the preceding actions assigns the same value as the cached values to the shared variables. As long as these conditions are fulfilled for a belief query \(\beta\), repeated querying of \(\beta\) returns the same substitutions for its involved variables, such that the query can be cached until one of the conditions is no longer met.
We will illustrate these conditions using the example in Figure 2. Consider the belief query in the guard of the first PG rule \((\text{enough\_fuel}(\text{Speed}) \text{ and } \text{dist}(D))\). The first condition states that the relevant part of the belief base should not be changed for this belief query. This will ensure that two identical belief queries provide the same result. If one of the belief updates \(\text{Forward}(Y)\) or \(\text{Backward}(Y)\) is successfully executed, it will update the value of \(\text{dist}(X)\) in the belief base and thus possibly change the result of the query in the guard of the rule, because this guard contains \(\text{dist}(D)\). Therefore, the query will have to be performed again and caching does not apply. The second condition states that the substitution of the variables that occur both in head and the guard of the rule should remain unchanged. This means that the substitutions of the variable \(\text{Speed}\) in the rule head should be that same as the previous query, which will ensure that the new belief query in the rule guard \(\text{enough\_fuel}(\text{Speed}) \text{ and } \text{dist}(D)\) is the same as previous query. The third condition does not apply.

We consider now the belief query action \(B(\text{D <= 0 and new\_speed}(\text{NewSpeed}))\). The first condition states again that the belief base should not change in a relevant way. Since no belief update action can update the value of the predicate \(\text{new\_speed}\) in the belief base, the result of this query cannot be affected by a belief update action. This means that the first condition is always fulfilled. The second condition does not apply. The third condition states that the variables shared with earlier actions (in this case the instantiation of \(D\)) should have the same instantiated value as in the previous execution of the query. In our case this means that the earlier substitution resulted from the execution of the belief query \(\text{enough\_fuel}(\text{Speed}) \text{ and } \text{dist}(D)\) should contain the same value instantiation for the variable \(D\) as in the current substitution for \(D\).

In order to verify whether the first condition holds it is necessary to determine which facts are relevant to belief queries. For this, we calculate the dependency sets for all belief queries in a program. The dependency set of a belief query contains all the atoms that can possibly affect the result of the query. Moreover, we calculate the relevant queries for a belief update action as follows: If the post-condition of a belief update action contains an atom that is in the dependency set of a belief query, this query will be added to the list of relevant queries for this belief update action. We build our idea of belief caching based on the relevant queries of the update actions. In particular, when the belief update action is invoked, a \textit{changed} flag will be set in its relevant queries. Thus, if the belief base has changed in a relevant way for a belief query, the \textit{changed} flag will be \textit{TRUE} for this query.

Note that it is possible to calculate the dependency sets of the queries and relevant queries for the belief update actions at compile-time because belief update actions and belief queries are static in 2APL, i.e., no new atoms will be added to the belief base at run-time. This means that this extension will be practically costless in terms of run-time performance. The extension we propose is two-fold. Firstly, the belief queries are extended with a cache to store previous substitutions, a \textit{changed} flag and a decision mechanism to apply caching. Secondly, the definition of a belief update is extended such that it is possible
to determine the relevant queries for each belief update. We will explain each extension in more detail in the next two sections.

### 3.1 Extended Belief Queries

Recall from Section 2.1 that both types of belief queries (as the guard of practical reasoning rules and as belief test actions) involve two substitutions $\tau_1$ and $\tau_2$. $\tau_1$ is the substitution that contains all variables that have been instantiated before the belief query, while $\tau_2$ is the substitution that contains all variables resulting from executing the query to the belief base.

To distinguish between belief queries that contain variables which are already instantiated, i.e. belief queries that contain variables that occur in $\tau_1$, and those that do not, we introduce the flag $\text{shared}$ for each belief query $\beta$ and use $\beta.\text{shared}$ to refer to this flag. This flag is set (i.e., it has the value $\text{true}$) when the code fragment before the query and the query itself share variables. In the case that the query occurs in the guard of a practical reasoning rule, this code fragment is the head of the practical reasoning rule. In the case that the query occurs in a belief query action, the code fragment is the actions that precedes the belief query action.

**Definition 1 (Shared belief query).** Let $H \leftarrow \beta | \pi$ be a practical reasoning rule and $\text{Var}(X)$ is the set of variables that occur in expression $X$. The flag shared of the belief query $\beta$ is set iff $H$ and $\beta$ share variables, i.e.:

$$\text{Var}(H) \cap \text{Var}(\beta) \neq \emptyset \implies \beta.\text{shared} = \text{true},$$

Moreover, let $\pi$ (the body of the practical reasoning rule) be a plan of the form $\pi'; \text{B}(\beta); \pi''$. Then, the flag shared of the belief query $\beta$ is set iff $\pi'$ and $\beta$ share variables, i.e.:

$$\text{Var}(\pi') \cap \text{Var}(\beta) \neq \emptyset \implies \beta.\text{shared} = \text{true},$$

For example, in Figure 2 the belief queries in both PG rules are shared because the variable $\text{Speed}$ occurs both in the rule head and rule guard. Similarly, the belief query actions in both rules are shared because the variable $\text{D}$ occurs both in the rule guard and the belief query action.

In order to perform caching, both substitutions $\tau_1$ and $\tau_2$ are stored for each query $\beta$ so that they can be re-used for the next query of $\beta$. Therefore we introduce for each query $\beta$ the substitutions $\tau_1$ and $\tau_2$. We cache these substitutions related to query $\beta$ and denote them by $\beta.\tau_1$ and $\beta.\tau_2$. We would like to emphasize that it may also be possible to store a history of substitutions $\tau_1$ and $\tau_2$ in order to reduce even more queries. This is particularly effective for when $\tau_1$ and $\tau_2$ share variables and $\tau_1$ changes, and the belief base does not change. Next, we introduce for each belief query $\beta$ the flag $\text{changed}$ that will be set whenever the belief base has been updated in relevant way, which means that caching does not apply and the query $\beta$ should be executed with respect to the belief base. The flag $\text{changed}$ associated with the belief query $\beta$ is denoted by $\beta.\text{changed}$. This
flag is set by belief update actions, which we will discuss in the next section. For now we simply assume that this flag always has the correct value.

Using these variables it is possible to define a decision mechanism that implements belief caching for the belief queries (Figure 3). If the relevant part of the belief base has been changed for the query $\beta$ (i.e., $\beta\.changed$ is TRUE), the belief query will always be executed. If query $\beta$ is shared and the cached substitution $\beta\.\tau_1$ is different from the current substitution $\tau_1$, the belief query $\beta$ is executed as well. The reason for this is that the cached substitution $\beta\.\tau_1$ applied to $\beta$ will result in a different query than applying the new substitution $\tau_1$ (which is different from $\beta\.\tau_1$) to $\beta$. After executing each belief query $\beta$, the corresponding flag $\beta\.changed$ is set to FALSE.

3.2 Extended Belief Updates

In this section we will define precisely how the caching flag $\beta\.changed$ is set for the belief queries. Recall from Section 2 that the only way in which the belief base can be updated is by belief updates. We will make use of dependency sets for queries, which we will now introduce. These dependency sets are defined for the belief base of 2APL, which is a general logic program.

**Definition 2 (Atom dependency [4]).** An atom $a$ depends on an atom $b$ in a general logic program $P$ iff (i) there exists a clause $C$ in $P$ such that $a$ is the head of $C$ and $b$ occurs in the body of $C$, or (ii) there exists a clause $C$ in $P$ such that $a$ is the head of $C$ and there is an atom $c$ in the body of $C$ that depends on $b$.

Note that the second condition of Definition 2 is recursive, meaning that an atom $a$ can depend on an atom $b$ via any number of clauses $C_1, C_2, ..., C_n$, given that $a$ occurs in the head of $C_1$, the head of each clause $C_i$ occurs in the body of the previous clause $C_{i-1}$ (given that $i > 1$) and $b$ occurs in the body of $C_n$.

---

3 When a leaf contains NOP, this means that no operation is performed.
Let $\pi(P)$ be the set of atoms occurring in the general logic program $P$. The atom dependencies in $P$ is a binary relation $R_{dpd} \subseteq \pi(P) \times \pi(P)$.

**Definition 3 (Dependency set [4]).** The dependency set for an atom $a$ in a general logic program $P$, denoted by $R^*_{dpd}(a)$, contains all atoms $b$ that $a$ depends on.

We can calculate the atom dependency set for an atom $a$ using the following two steps, which are a reformulation of the conditions given in Definition 2: 1) Add the atom $a$ in the atom dependency set, 2) Add all atoms occurring in the body of clauses in which atoms in the dependency set occur in the head. Step (2) is repeated until this set does no longer grow. We can straightforwardly extend the definition of an atom dependency set for a belief query.

**Definition 4 (Query dependency set).** The query dependency set for a query $\beta$ to a general logic program $P$, denoted by $R^*_{dpd}(\beta)$, is the union of the atom dependency set of each atom that occurs in $\beta$.

$$R^*_{dpd}(\beta) = \bigcup_{a \in \beta} R^*_{dpd}(a).$$

Suppose a query $\beta$ is executed at deliberation cycles $C_1$ and $C_2$ and that the previous substitution $\beta.\tau_1$ is equal to the current substitution $\tau_1$. The only way in which the result of this query can change is if the substitution in $C_2$ of a variable $X$ that occurs in an atom in the dependency set of $\beta$ is different from the substitution of $X$ in $C_1$. So, if an atom that occurs in the post-condition of a belief update is a member of the query dependency set of a belief query, then that belief update action can affect the substitution of such a variable $X$.

Consider for instance the belief query $\text{enough fuel}(\text{Speed})$ and $\text{dist}(\text{D})$ that occurs in the guard of the first PG rule in the example program in Figure 2. According to Definition 4, the query dependency set for a query $\beta$ is the union of the atom dependency set of each atom that occurs in this query. In this case, this is the union of the atom dependency sets of the atoms $\text{enough fuel}$ and $\text{dist}$. This is calculated using the belief base:

```
dist(50).
default_fuel(X) :- X is int(random(10)).
fuel(1000).
enough_fuel(X) :- fuel(Y), X =< Y.
```

We calculate the atom dependency set using the algorithm that we stated directly after Definition 3. First add $\text{enough fuel}$ to the set. Then add all atoms occurring in the body of rules in which $\text{enough fuel}$ occurs in the head. This means that $\text{fuel}$ is added to the set, because the last rule in the logic program fulfills this condition. The atom dependency set is now $\{\text{enough fuel}, \text{fuel}\}$. After this step, adding atoms that occur in the body of rules in which $\text{enough fuel}$ or $\text{fuel}$ occur in the head does not increase the size of the set,
which means that the atom dependency set is complete. Because the atom \texttt{dist}
does not occur in any clause where there are atoms in the body, the atom
dependency set of this atom is simply \{\texttt{dist}\}. This means that the query dependency
set of \texttt{enough\_fuel(Speed) and dist(D)} is \{\texttt{enough\_fuel, fuel, dist}\}.

Now, if an atom that occurs in the post-condition of a belief update is a
member of this set as well, it can affect the result of this query. Recall that the
belief updates of Figure 2 are:

\{ \texttt{dist(X) and fuel(F)} \} Forward(Y) \{ \texttt{not dist(X), not fuel(F), dist(X - Y), fuel(F - Y)} \}
\{ \texttt{dist(X) and fuel(F)} \} Backward(Y) \{ \texttt{not dist(X), not fuel(F), dist(X + Y), fuel(F - Y)} \}

Since both belief updates contain the atom \texttt{dist} and the atom \texttt{fuel} and both
these atoms occurs in the query dependency set of the belief query \texttt{enough\_fuel(Speed)
and dist(D)}, both belief updates are \textit{relevant} for this query. We make the con-
cept of belief query relevance more clear in the following definition.

\textbf{Definition 5 (Belief query relevance).} A belief update \( \alpha \) is relevant for a
belief query \( \beta \) if an atom \( a \) occurs both in the postcondition of \( \alpha \) and in the
dependency set of \( \beta \).

All relevant queries for a belief update are put in a set and activated whenever
the belief update action is executed by setting the \textit{changed} flag of these queries
to \texttt{true}.

\textbf{Definition 6 (Extended belief update).} We add to each belief update \( \alpha \) a
set \texttt{relevantQueries} containing belief queries and execute the algorithm depicted
in Algorithm 1 at compile-time. We also add for each belief update the algorithm
depicted in Algorithm 2 that is executed when the belief update action is executed.
Call the resulting belief update an extended belief update.

\begin{algorithm}
\caption{Collect relevant queries for each belief update action.}
\begin{algorithmic}[1]
\Procedure{collectRelevantQueries}{ }
\For{all beliefupdate \( \alpha \)}
\For{all query \( \beta \)}
\If{\( \exists p : p \in R^\text{upd}_{\alpha}(\beta) \land p \in \text{postcondition}(\alpha) \)}
\State \( \alpha.\text{relevantQueries}.\text{put}(\beta) \)
\EndIf
\EndFor
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

\subsection{Abstract performance model}

The abstract performance model for logic-based agent programming languages,
as proposed in [1], can be used in order to measure the effect of belief caching.
According to this model, the three steps in the deliberation cycle of a 2APL
agent can be mapped onto two kinds of knowledge representation functionality:
Algorithm 2 Reset caching for relevant queries for each belief update action.

1: procedure setRelevantQueries (α)
2:     for query β in α.relevantQueries do
3:         β.changed ← TRUE
4:     end for
5: end procedure

the query phase and the update phase. Together, they constitute an update cycle (Figure 4). The query phase is a phase in which one or more belief queries are performed, and in which no belief updates take place. As soon as a single belief update occurs, the model switches to the update phase. It will remain in the update phase until a single belief query takes place. The belief caching mechanism proposed in [1], which we will call the original caching mechanism, is to cache the queries within one query phase by making use of a hash table that contains all queries that have been performed in this query phase. This will ensure that the belief base has not been changed, simply because no belief update has occurred. The complete cache is cleared as soon as the model switches to the update phase, i.e. a single belief update takes place.

Fig. 4: The abstract performance model [1]

Our implementation is more fine-grained, though, since it refines the general update cycle of [1] to an update cycle for each individual belief query. This means that each single belief query goes through the update cycle of Figure 4. Therefore the number of update cycles for individual queries are independent, while in the case of [1] a single belief update will reset the cache of all queries. This means that our proposal will lead to more belief caching in the case that the update cycles of the individual belief queries are not identical, a situation which frequently occurs.

4 Experimentation

4.1 Experimental Setup

We have analyzed the working of belief caching using a benchmarking tool that was developed for this work. We have tested belief caching for three increasingly realistic programs.\textsuperscript{4}

\textsuperscript{4} The sources for the used programs can be downloaded from http://www.students.science.uu.nl/~3714314/2apl_beliefcaching_examples.rar
The first program (driver) has been developed to demonstrate the working of belief caching specifically. The code of this program is almost identical to Figure 2, except that the body of the Prolog rule enough_fuel has been replaced by a computationally heavy calculation involving integers.

The second program (storage) has been written for this task as well but is more realistic. It consists of a multi-agent system with 10 different agents that each can store items in a storage list. Agents will attempt to keep their items stock constant while they receive items from the environment.

The last program (marketplace) is an existing and more sophisticated version of a multi-agent system in which agents have items that they can sell, and have items that they want to buy. Agents can bid for items they desire and sell an item when a bid of another agent meets their demands.

We have compared the results between 2APL with and without belief caching. We use ”2APL” to refer to 2APL with no belief caching, and ”2APL*” to refer to 2APL with belief caching. All experiments have been performed on a 2.4GHz Intel Core i5, 6 GB 667 MHz DDR3, running Windows 7 and Java 1.6.

When showing the benchmarking results, we use \(d\) to denote the number of deliberation steps, \(Q_b\) for belief queries, \(U_b\) for belief updates, \(C_{PC}\) for PG rule calls, \(C_{PC}\) for PC rules calls, \(C_{PR}\) for PR rule calls, and \(B\) for the run-time of the program, which we will also refer to as the benchmarking time.

4.2 Results

Driver program We plot the number of deliberation steps while we increase the run-time of the program (the benchmarking time) (Figure 5a). The asymptotic behavior is due to the fact that initially there exist no cache for the beliefs, but this will be less and less relevant as the benchmarking time increases. The asymptote of 2APL lies around 400 deliberation steps per second, and the one of 2APL* around 4700, which is over ten times as much.

![Graph showing deliberation steps per second and average processing time of a PG rule.](image)

(a) Deliberation steps per second. (b) Average processing time of a PG rule.

Fig. 5: (Driver) Results for increasing benchmarking time.
The only rules that are being used in the **driver** program are PG rules. Therefore, it is of interest to see whether the PG rules are being processed faster because of belief caching. When we plot these values for increasing benchmarking times (Figure 5b), we see that 2APL* processes PG rules much faster than 2APL. Where 2APL has an asymptote at around 7 ms per call, 2APL* has one of around 0.5 ms, which is about 14 times faster.

![Fig. 6: (Driver) Total number of calls for all operations (B=240s)](image)

The reason why PG rules are being processed much faster in 2APL* is because less time is spent on doing belief queries. For completeness, the graph showing the total number of calls in 240 seconds for all operations is depicted in Figure 6, which shows that indeed the number of calls for all operations have increased drastically for 2APL*.

**Storage program** Initial experiments showed that the amsymptote of this program lies around a benchmarking time of 200 seconds. We will therefore limit ourselves to this benchmarking time.

In Figure 7a we have depicted the number of calls for the different operations at a benchmark time of 200 seconds. As we can see, the number of deliberation steps has improved with a factor of about 4 for 2APL*, which is significant. The number of belief queries has remained more of less constant, but since much more deliberation steps have been executed, the number of belief queries per deliberation step has decreased a lot. This is shown more clearly in Figure 7b, where we see that the belief queries take up much less processing time in the case of 2APL*.

**Storage** uses many operations on lists that might not have changed, which means that these operations are omitted when using belief caching. Because the multi-agent system consists of 10 agents, the difference is rather big. We have deliberately used multiple agents to show that difference can become very significant when scaling up the multi-agent system.

**Marketplace** The last program that we test contains only very simple belief queries. The question that we would like to answer is whether such a program could also be improved using belief caching. As we see in Figure 8a, the number
of deliberation steps increases slightly when using 2APL*, while the number of belief queries decreases with half. This makes sense, because while we save many belief queries, there is not much increase in run-time because the queries are very simple and not time consuming. This becomes more clear in Figure 8b, where the processing time of the different operations is shown. As we can see, the operation time of the belief queries is very small and this does not affect the efficiency of the program greatly.

5 Conclusion

We have implemented belief caching into 2APL and showed that it extends the abstract performance model of [1]. Instead of single-cycle caching, our implementation keeps track of an update cycle for each individual belief query. We have implemented belief caching into the latest version of 2APL.

The benchmarking results show that belief caching can optimize a 2APL program significantly, because it is an effective way to reduce the number of
belief queries. To what extent this decrease will contribute to an increase in deliberation speed depends on the complexity of the belief queries.

Our most important contribution is that the implementation will never lead to a worse performance, because the dependencies between the belief updates and the belief queries can be calculated at compile-time.

Logic-based agent programming language are based on a combination of imperative programming with logic-based knowledge bases. Because this approach is relatively new, there has not been much research dedicated towards the optimization of the communication between these two formalisms. Our approach has shown that it can be very beneficial to optimize this. We therefore see it as a first step towards increasing the efficiency of logic-based agent programming languages so that they will become better applicable to practical domains.

We plan to continue our optimization work on 2APL by building goal caching mechanism as well as a mechanism that decreases the set of applicable practical reasoning rules. It should be noted that the current 2APL interpreter checks at each deliberation cycle which practical reasoning rule is applicable. This is done by checking the head and guard of the rules which requires queries to belief, goal, and event bases. Any mechanism that keeps track of non-applicable rules may reduce the number of applicable practical reasoning rules and thus the number of time consuming queries.

We believe that our caching mechanism is not limited to 2APL. It can be implemented into logic-based agent programming languages such as Jason [2], GOAL [5], or other multi-agent programming languages that combine logic-based knowledge bases with imperative programming (see [7] for an overview), as long as the set of plan rules do not change at run-time. We leave this issue for further research. Moreover, we believe that the proposed caching mechanism does not change the behavior of 2APL interpreter. In future work, we aim at providing proofs for the soundness and completeness of our caching mechanism.

References

Abstract. Testing multi-agent systems is a challenge, since by definition such systems are distributed, and are able to exhibit autonomous and flexible behaviour. One specific challenge in testing agent programs is developing a collection of tests (a “test suite”) that is adequate for testing a given agent program. In order to develop an adequate test suite, it is clearly important to be able to assess the adequacy of a given test suite. A well-established technique for assessing this is the use of mutation testing, where mutation operators are used to generate variants (“mutants”) of a given program, and a test suite is then assessed in terms of its ability to detect (“kill”) the mutants. However, work on mutation testing has focussed largely on the mutation of procedural and object-oriented languages. This paper is the first to propose a set of mutation operators for a cognitive agent-oriented programming language, specifically GOAL. Our mutation operators are systematically derived, and are also guided by an exploration of the bugs found in a collection of undergraduate programming assignments written in GOAL. In fact, in exploring these programs we also provide an additional contribution: evidence of the extent to which the two foundational hypotheses of mutation testing hold for GOAL programs.

Keywords: Mutation Testing, Agent-Oriented Programming Languages, GOAL

1 Introduction

Testing multi-agent systems (MAS) is a challenge, since by definition such systems are distributed, and are able to exhibit autonomous and flexible behaviour. For example, Munroe et al. note that “However, the task [validation] proved challenging for several reasons. First, agent-based systems explore realms of behaviour outside people’s expectations and often yield surprises…” [12, Section 3.7.2]. Similarly, Pěchouček and Mařík [16, Page 413] note that: “Although the agent system performed very well in all the tests, to release the system for production would require testing all the steel recipes with all possible configurations of cooling boxes”.

There has been work on testing of multi-agent systems, especially in the last 4-5 years. Most of this work has focussed on tool support for executing (manually defined) tests (e.g. [2, 3]). However, some work has investigated test generation based on design models [20], ontologies [13], or using evolutionary techniques [14]. Space precludes a detailed review of testing, and we refer the reader to [19, Section 8.1] for a review of work on testing and debugging MAS. Overall, the conclusion of this review was that “testing of agent based systems is an area where there is a need for substantial additional work” [19, Section 8.1].
Given a collection of tests (a “test suite”), a key question when testing an agent system is to what extent is the test suite adequate? A test suite is adequate to the extent that it is able to distinguish between a correct and an incorrect program. In developing an adequate test suite, it is obviously useful to be able to assess the adequacy of the test suite. This assessment can assist a tester in detecting when a test suite is not sufficient and needs to be refined or extended. It can also guide a tester in knowing when to stop adding test cases.

So far, work on assessing the adequacy of test cases (e.g. [9, 11, 18]) has only considered the use of various coverage metrics to assess test suite adequacy. However, although coverage is necessary, it is not sufficient. Knowing that a test suite covers a certain portion of a program simply indicates that parts of the program were executed by the tests. It doesn’t allow us to draw conclusions about whether these parts of the program were tested in a way that allows errors in the program to be detected, i.e. to distinguish between correct and incorrect programs.

An alternative, well-established, technique for assessing test suite adequacy is mutation testing [6] (see Section 2.1). Mutation testing directly assesses the ability of a test suite to distinguish between different programs, and is considered a more powerful and discerning metric than coverage: e.g. “If your tests are adequate with respect to some other adequacy criteria … then chances are that these are not adequate with respect to most criteria offered by program mutation” [10, Page 503].

Most work on mutation testing of programs has focussed on programs in procedural and object-oriented languages [6, Figure 5]. There has been a (very) small amount of work on applying mutation to agents [17, 1]. However, this work has not considered mutating agent programs written in a cognitive agent-oriented programming language.

In this paper we propose a set of mutation operators for the cognitive agent-oriented programming language GOAL (see Section 2.2 for a brief introduction to the language). Although we derive mutation operators for a specific language, the process by which the operators are derived is generic, and can easily be applied to other agent-oriented programming languages (see Section 6).

In deriving our mutation operators we are guided by an exploration of actual bugs in GOAL programs. We want mutation operators to generate “realistic” bugs, and we assess this by considering a collection of GOAL programs (written by undergraduate students at another university). Section 4 compares the bugs that exist in these programs against the sorts of bugs that our mutation operators generate, and uses the results to guide the selection of mutation operators. In fact, the results of this assessment of bugs also forms an additional contribution, in that it provides evidence of the extent to which the two foundational hypotheses of mutation testing hold for GOAL programs.

The remainder of this paper is structured as follows. We begin by briefly reviewing mutation testing (Section 2.1) and introducing the GOAL programming language. We then present our mutation operators in Section 3. Section 4 looks at a collection of GOAL programs and considers what bugs they contain. We then (Section 5) describe an implementation of the mutation operators, and report the number of mutants generated by different operators for a number of example GOAL programs. Finally, we conclude with a discussion, including future work (Section 6).
2 Background

2.1 Mutation Testing

We now very briefly introduce the key ideas of mutation testing, which is a long-established field, going back to the 70s. For a detailed introduction to mutation testing see Chapter 7 of [10], and for a recent review of the field see Jia & Harman [6]. In a nutshell, mutation testing assesses the adequacy of a test suite by generating variants (“mutants”) of the program being tested, and assessing to what extent the test suite is able to distinguish the original program from its mutants (termed “killing the mutant”). Given a test suite, a program \( P \) written in a programming language, and a set of mutation operators for that programming language, the process of mutation testing is as follows:

1. Execute the program \( P \) against all tests in the test suite, recording the results;
2. Use the mutation operators to generate a set of mutant programs \( P_1 \ldots P_n \) from \( P \) (where each \( P_i \) is the result of applying a single mutation);
3. Test each mutant \( P_i \) against the tests in the test suite;
4. Each mutant that behaves differently to the original program is flagged as having been “killed”\(^1\);
5. The adequacy of the test suite is \( D/n \) where \( D \) is the number of killed mutants and \( n \) is the number of mutants. A quality score of 1 (highest) is good, and 0 is bad.

The mutants are generated using mutation operators: rules that take a program and modify it, yielding a syntactically valid variant. The key challenge in developing a mutation testing scheme is the definition of a good set of mutation operators for the programming language used. A set of mutation operators is good to the extent that it (1) generates errors that are realistic; and (2) does so without generating a huge number of mutants.

Mutation testing rests on two foundational hypotheses [6]. The first is the competent programmer hypothesis, which states that programmers tend to develop programs that are close to being correct. This hypothesis is important in that a consequent of it is that a simple syntactic mutation is a good approximation of the faults created by competent programmers. In other words, the competent programmer hypothesis is what justifies the use of simple syntactical mutations as proxies for real bugs. The second foundational hypothesis is the coupling effect hypothesis. This proposes that a test suite that can find the simple faults in a program, will also find a high proportion of the program’s complex faults [15]. This hypothesis justifies the generation of mutants by the application of a single mutation operator instance, rather than having to consider the application of multiple mutation operators to generate a mutant.

In Section 4 we consider a collection of GOAL programs and assess to what extent these two foundational hypotheses hold. Although there is empirical evidence to support both these hypotheses for procedural programs, this paper is the first to consider evidence for these hypotheses in the context of agent systems.

\(^1\) Some mutants may be equivalent in behaviour to the original program (“equivalent mutants”), and, since program equivalence is undecidable, identifying and removing these mutants is a manual and partial process. This is a standard problem in the field of mutation testing but there is evidence that most equivalent mutants are actually fairly easy to detect.
2.2 GOAL

This section briefly introduces GOAL (Goal Oriented Agent Language); for further details we refer the reader to the existing literature [5, 4]. A Multiagent System in GOAL is defined using a configuration file that specifies the environment, configuration options, and a number of GOAL agents, each with a GOAL program. A GOAL agent program consists of five components: domain knowledge (e.g. Prolog rules), initial beliefs, initial goals, action definitions, and a program. Note that both the domain knowledge and the beliefs are specified using a knowledge representation language which can be varied (the GOAL implementation uses SWI-Prolog). In this paper we focus on the program component, both because it corresponds most closely to other agent-oriented programming languages (AOPLs), and because that is usually where the complexity of the agent is, and where errors are made (see Section 4).

GOAL programs are built out of actions and mental state conditions. Actions in GOAL are either user-defined (pre and post condition), or are one of the five built-in actions that insert or delete beliefs, adopt or drop goals, or send a message. GOAL also defines an achievement goal $a\text{-goal}(\phi) \equiv \text{goal}(\phi) \land \neg \text{bel}(\phi)$ and an achieved goal $\text{goal-a}(\phi) \equiv \text{goal}(\phi) \land \text{bel}(\phi)$. A mental state condition (MSC) in GOAL is built out of conditions over the agent’s beliefs and its goals. A GOAL program definition then, in essence, consists of a sequence of rules of the form $\text{if MSC then action}_1 + \ldots + \text{action}_n$ where the actions are performed in order, and there can be at most one user-defined action.

These rules are actually placed within modules. However, in this paper we do not consider the mutation of modules, since the module construct is unique to GOAL, and was not used in the programs we considered (see Section 4). The grammar in Figure 1 summarises the subset of the language that we focus on. It thus differs from the original grammar given by Hindriks [5] in that it omits modules. It also differs in a couple of places where it has been changed to match what the implementation supports (specifically for drop(\phi) the \phi must not contain negations; and in fact in mentalatoms belief conditions can actually contain disjunctions).

A rule “if condition then action” is applicable if the condition holds, and is enabled if the actions’ preconditions are met. Applicable and enabled rules are options. The execution cycle consists of the following steps:

1. Clear previous cycle’s percepts.
2. Update percepts by executing all options (i.e. enabled rules) in the distinguished event module.
3. Focus on the main module: compute the options, select one (by default rules are evaluated in linear order and the first option is selected), and perform its actions.
4. Update goals by dropping goals that are believed to hold.

Compared with other cognitive agent programming languages, GOAL’s most distinctive (relevant) features are: (a) The limitation that an action rule can only result in a sequence of actions, rather than a mixture of actions and subgoals; and (b) The lack of a trigger condition. This makes GOAL action rules more general, in that a rule doesn’t

\footnote{There is also a form “forall MSC do actions” used in the percept processing module.}
program ::= actionrule+
    actionrule ::= if mentalstatecond then actioncombo
        | forall mentalstatecond do actioncombo

    mentalstatecond ::= mentalliteral { , mentalliteral }∗
    mentalliteral ::= true | mentalatom | not( mentalatom )
    mentalatom ::= bel( litconj ) | goal( litconj )

    actioncombo ::= action { + action }∗
    action ::= user-def-action | built-in-action | communication
    user-def-action ::= id[parameters]
    built-in-action ::= insert( litconj ) | delete( litconj )
        | adopt( poslitconj ) | drop( poslitconj )
    communication ::= send( id , poslitconj )

    poslitconj ::= atom { , atom }∗.
    litdisj ::= litconj { ; litconj }∗ .
    litconj ::= literal { , literal }∗.
    literal ::= atom | not( atom )

    atom ::= predicate[parameters] | ( litdisj )
    parameters ::= ( term { , term }∗ )

Fig. 1. GOAL Agent Program syntax (adapted from [5]): term is a legal term and id is an identifier.

require a particular trigger. However, it also means that a rule can be applied repeatedly: in, say, Jason a rule of the form +!goal : context ← planBody can be applied (if the context is true) to deal with the creation of a goal. However, the rule will not be applied subsequently unless the goal is re-posted. By contrast, in GOAL a rule of the form “if goal(goal) then actions” can be applied repeatedly, as long as goal remains a goal of the agent.

3 GOAL Mutation Operators

"the design of mutation operators is as much of an art as it is science." [10, Page 530].

In deriving a set of mutation operators for GOAL we follow the approach of Kim et al. [8] and derive mutation operators based on HAZOP and the syntax of the language. HAZOP (Hazard and Operability Study) is a technique for identifying hazards in systems by considering each element of the system and applying “guide words” such as NONE, MORE, LESS, PART OF, AS WELL AS, or OTHER THAN. For example, in a chemical processing system, engineers might consider what hazard exists if a certain pipe carries MORE chemical than it should, or if there is a contaminant ("AS WELL
Kim et al. applied this idea to generating mutation operators by applying these guide words to the syntax of Java. For example, when considering a method invocation, the guide word OTHER THAN suggests that the designer consider the possibility that a different method to the intended one is invoked. This then leads directly to the definition of a mutation operator that rewrites a method invocation by changing the method name. Figure 2 shows their interpretation of the HAZOP guide words (note that some of the guide words, such as those to do with scope, or quantitative changes, are not applicable to GOAL, and so have been left out of the figure).

<table>
<thead>
<tr>
<th>Guide Words</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO/NONE</td>
<td>No part of the intention is achieved. No use of syntactic components.</td>
</tr>
<tr>
<td>AS WELL AS</td>
<td>Specific design intent is achieved but with additional results</td>
</tr>
<tr>
<td>PART OF</td>
<td>Only some of the intention is achieved, incomplete</td>
</tr>
<tr>
<td>REVERSE</td>
<td>Reverse flow - flow of information in wrong direction ... negation of condition</td>
</tr>
<tr>
<td>OTHER THAN</td>
<td>A result other than the original intention is achieved, complete but incorrect</td>
</tr>
</tbody>
</table>

Fig. 2. HAZOP guide words and their interpretation for software (copied from [8])

In deriving mutation operators for GOAL we actually go through two stages. We firstly apply HAZOP to abstract syntactical classes in order to develop generic mutation schemas (Figure 3). These are generic in that they are applicable to a wide range of programming languages. We then apply these schemas to the GOAL syntax to generate specific mutation rules for GOAL (Figure 4). The advantage of doing the derivation in two stages is that we can then more easily derive mutation operators for other AOPLs by applying the generic schemas.

In deriving our generic schemas we consider three generic syntactical types: a sequence of elements, a (binary) operator that has two sub-elements, and a unary operator. For each of these generic syntactical types we consider what mutations are suggested by the HAZOP guide words, which gives a set of generic mutation schemas for that syntactical type. In addition, there is also a generic schema, suggested by the NO/NONE HAZOP guide word, that can be applied to delete (“drop”) any syntactical type. Formally we write drop:$x \rightsquigarrow \epsilon$ to capture this: the “drop:” is a label, $x$ is a variable for a syntactical element, the arrow “$\rightsquigarrow$” indicates a mutation, and $\epsilon$ denotes the empty syntactical construct of the appropriate form (e.g. empty sequence, “True” Boolean value).

Consider now a sequence of elements ($x_1 \ldots x_n$). The HAZOP guide word PART OF suggests that we consider removing an element in the sequence (“drop1” in Figure 3). The OTHER THAN guide word suggests changing part of the sequence, specifically, we select an element, and replace it with a variant (derived using other appropriate mutation operators; “mut1”). The REVERSE suggests changing the order of the sequence. However, in general reversing a sequence of syntactic elements doesn’t make much sense, and instead we propose a rule to swap two adjacent elements in the sequence (“seqswap”): note that we choose to only swap adjacent elements in order to avoid generating a large number of possible mutants (but see the discussion of program:seqtop and seqbot later in this section). The AS WELL AS guide word suggests that we add an item to the sequence. However, we prefer to avoid adding things because
this raises the issue of what to add? If we add, say, a new rule to a GOAL program, what rule should we add? It is possible to define a way of creating a new rule from existing fragments in the program, but this tends to result in a very large number of possible mutations. The NO/NONE guide word has already been handled by a rule that applies to all syntactical types, including sequences.

Consider now a binary operator (notation: $x \oplus y$ or $x \otimes y$, where we assume that $\oplus$ and $\otimes$ are different). Using similar reasoning, we are inspired by PART OF to consider dropping either the left or right element (“dropL”, “dropR”); by REVERSE to swap the elements (“swap2”); and by OTHER THAN to change either the operator (“op2”) or to mutate one of the elements (“mutL”, “mutR”).

Finally, consider a unary operator (notation: □$x$ or ♦$x$, assuming □ ≠ ♦). Using similar reasoning, we are inspired by the PART OF guide word to delete the operator (“delop”); by OTHER THAN to change the operator (“op1”) or mutate the component (“mut”). We also can add an operator (“addop”) which can be seen as inspired by the “negation of condition” interpretation of the REVERSE guide word (e.g. $F \iff \neg F$).

Figure 3 shows the resulting generic mutation schemas. Recall that each rule is of the form “keyword : $x \rightsquigarrow y$”. We also employ a convention that where we have $p(x) \rightsquigarrow p(x')$, there is an implied “if $x \rightsquigarrow x'$”. In other words, the mut1 rule, for instance, is really shorthand for “mut1: $x_1 \ldots x_j \ldots x_n \rightsquigarrow x_1 \ldots x'_j \ldots x_n$ if $x_j \rightsquigarrow x'_j$.”

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drop</td>
<td>$x \rightsquigarrow \epsilon$</td>
</tr>
<tr>
<td>seqswap</td>
<td>$x_1 \ldots x_j x_{j+1} \ldots x_n \rightsquigarrow x_1 \ldots x_{j+1} x_j \ldots x_n$</td>
</tr>
<tr>
<td>mut1</td>
<td>$x_1 \ldots x_j \ldots x_n \rightsquigarrow x_1 \ldots x'_j \ldots x_n$</td>
</tr>
<tr>
<td>dropL</td>
<td>$x \oplus y \rightsquigarrow y$</td>
</tr>
<tr>
<td>dropR</td>
<td>$x \oplus y \rightsquigarrow x$</td>
</tr>
<tr>
<td>swap2</td>
<td>$x \oplus y \rightsquigarrow y \oplus x$</td>
</tr>
<tr>
<td>op2</td>
<td>$x \oplus y \rightsquigarrow x \otimes y$</td>
</tr>
<tr>
<td>mutL</td>
<td>$x \oplus y \rightsquigarrow x' \oplus y$</td>
</tr>
<tr>
<td>mutR</td>
<td>$x \oplus y \rightsquigarrow x \oplus y'$</td>
</tr>
<tr>
<td>addop</td>
<td>$x \rightsquigarrow \diamond x$</td>
</tr>
<tr>
<td>delop</td>
<td>$\diamond x \rightsquigarrow x$</td>
</tr>
<tr>
<td>op1</td>
<td>$\diamond x \rightsquigarrow \Box x$</td>
</tr>
<tr>
<td>mut</td>
<td>$\diamond x \rightsquigarrow \diamond x'$</td>
</tr>
</tbody>
</table>

Fig. 3. Generic Mutation Schemas

The second step is to apply these generic mutation schemas to the GOAL syntax in order to derive a set of mutation operators specific to GOAL. In doing so, we sometimes leave out rules that don’t make sense. For example, when a binary operator is commutative, it doesn’t make sense to mutate by swapping its arguments (“swap2”). We now proceed to consider in turn each syntactical element type in GOAL and consider how the generic mutation schemas apply to it.

We begin by considering a GOAL program. This is a sequence of action rules, and therefore the relevant generic mutation schemas are those for a sequence (seqswap, mut1, drop1), as well as the universal “drop” schema. In this case, it doesn’t make sense to drop the whole program, so we have three rules (labelled in Figure 4 “program:seqswap”, “program:mut1” and “program:drop1”). In fact, in our exploration of bugs in example GOAL programs, we also found that the limitation to only swap adjacent rules in a program was too strong: there were a number of cases where bugs corresponded to other sorts of changes to the order of rules in a program. Although we do not want to introduce a mutation operator to allow arbitrary reorderings of the rules.
in a program, we do propose a compromise that allows many of the bugs seen to be
generated by our mutation rules, whilst not increasing the number of possible mutants
too much. This compromise is to add mutation operators that allows a single rule in the
program to be moved to the start (“program:seqtop”) or end (“program:seqbot”) of the
program.

Next we consider a GOAL action rule (abbreviated AR). An action rule is effec-
tively a binary connective with two sub-components, and hence the generic schemas
for binary connectives apply (i.e. dropL, dropR, swap2, op2, mutL, mutR, as well as
drop). However, for an AR the components cannot be deleted, since a rule must have
both a condition and actions (although an MSC could be replaced with “true”), and they
cannot be swapped, so we only have rules for op2, mutL, and mutR. For op2 we also
consider replacing “if MSC then AC” with “forall MSC do AC” (and vice versa), but
only in the context of the percept processing module. The universal “drop” rule isn’t
needed for actionrules, since actionrules only occur within a sequence, and we already
have a rule to delete an element in the sequence. Note that op2 has two instances, and
that it is fairly specific to GOAL: other AOPLs don’t deal with percepts in the same way.

A mentalstatecond (MSC) is also a sequence. Here it does make sense to also con-
sider the overall drop rule, dropping the whole MSC, as well as the usual mut1 and
drop1 rules. However, in fact the result of dropping an MSC completely is rarely a valid
GOAL program: GOAL requires that variables appearing in the actions of a rule also
appear in that rule’s condition. Since this requirement only holds for a “true” condition
when the action list has no variables, the MSC:drop rule is unlikely to ever be applica-
table (see Section 5). Note that we only consider mutation by dropping an MSC if it has
more than one element (otherwise the same effect is achieved by the ML:drop rule).
Finally, since order in an MSC doesn’t matter, swapping doesn’t make sense.

A mentalliteral (ML), as defined in the GOAL syntax, is an optional unary operator
(“not”) that is applied to a mental atom (which itself is either a bel or a goal operator
applied to a litconj). We can mutate a mentalliteral by dropping it completely. We can
also add or remove a “not” (“addop”, “delop”), or we can mutate the mental atom.
Mutating a mental atom (MA) can be done by either changing a bel to a goal (or vice
versa), or by mutating the literal conjunction. Note that we cannot mutate the “not” into
another operator, since there is no alternative operator.

An actioncombo (AC) is a sequence of actions. It cannot be entirely deleted. How-
ever, we can mutate an individual action or drop one. Note that the swap rule doesn’t
really make sense: although GOAL specifies that actions in an actioncombo are exe-
cuted sequentially, it would in fact be non-idiomatic to have a sequence of actions that
is order dependent.

An action (A) is either user defined (“id[parameters]”) or is one of the five built-in
actions: insert, delete, adopt, drop, send. Dropping an action completely is already cov-
ered by the rule AC:drop1, so we only consider mutating the parameters of the action,
or the action type. In mutating the action type we exclude changing a belief operation to
a goal operation and vice versa, since this doesn’t make sense, and is unlikely to yield
a sensible mutant. When mutating a message, we can mutate the message content, or
the recipient. Finally, we can mutate a user-defined action by mutating either the id (by
replacing it with a different user defined action or by mutating the parameters. In replac-
ing a user-defined action with a different user-defined action we need to ensure that the

two actions have the same number of parameters (which we term being “compatible”).

This condition also applies when mutating atoms by changing their predicate.

We did observe that some programs had “typos” (e.g having “at” instead of “at-

Block”) in predicates. However, we did not introduce a mutation operator to create

such typos for the simple reason that this operator would be redundant. Consider, for

example, replacing the action “delete(p)” with “delete(typo)”. This is actually equiv-

alent to just deleting the action. Similarly, replacing “bel(p)” with “bel(typo)” in an

MSC is equivalent to replacing it with “false”, i.e. with deleting the rule; and having

“adopt(typo)” is equivalent to a null action since the goal won’t have rules that handle

it, so won’t have any effect.

A poslitconj (PLC) and a litconj (LC) are both sequences (respectively of Atoms

“At” or Literals “Lit”), so we can remove an element of the sequence or mutate an ele-

ment of the sequence. However, since they are conjunctions, order doesn’t matter, and

hence swapping elements doesn’t make sense. A Literal is an optional unary connective

(not) applied to an Atom, and hence can be mutated by adding or removing a negation,

or by mutating the atom. Mutating an atom can be done by mutating the predicate (re-

placing it with another predicate found in the program), or by mutating the parameters.

Parameters are a sequence of terms (we abbreviate term to t), but we do not want to

change the length of the sequence, hence can only mutate individual terms. However,

swapping is a reasonable mutation.

Mutating a list (of the form [t1 | t2]) is done similarly to any other binary connective,

yielding the following rules (for space reasons, these are not shown in Figure 4):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example 1</th>
<th>Example 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>termlist:drop1</td>
<td>[A</td>
<td>As] \sim A</td>
</tr>
<tr>
<td>termlist:seqswap</td>
<td>[A</td>
<td>As] \sim [As</td>
</tr>
<tr>
<td>termlist:mut</td>
<td>[A</td>
<td>As] \sim [A'</td>
</tr>
</tbody>
</table>

Finally, we consider the mutation of terms (excluding lists). A term is of the form

f(t_1, \ldots, t_n) and, viewed as a sequence of arguments, can be mutated by dropping a

sub-term (“term:drop1”), mutating a sub-term, or swapping adjacent sub-terms. There

is one special case: equality. For the term t_1 = t_2 it does not make sense to drop either

sub-term, nor to swap the sub-terms.

Figure 4 shows the collected mutation operators for GOAL. Recall that by conven-
tion where we have p(x) \sim p(x'), there is an implied “if x \sim x'”. We also assume that

there are implicit checks for syntactic elements being of the correct type. For instance,
in the rule ML:addop, the element MA must be a MentalAtom, and hence cannot be

“true” or “not(MA)”. There is also a constraint: for those rules that involve an element

j + 1 (i.e. the swap and drop1 rules) we have 1 \leq j and j + 1 \leq n, and hence that n \geq 2

(so we don’t drop the last element in a list). For other rules we have 1 \leq j \leq n. Finally,
the program:mut rule has an additional condition, discussed above, that all variables
appearing in the actions also appear in the rule’s condition.

These rules have the property that when applied to a valid GOAL program, they
result in another valid program, since they always replace a syntactical element of a
certain type (e.g. MSC) with another valid syntactical element of the same type (see
also Section 5). Figure 5 shows an example GOAL rule and its mutations (generated by
the implementation described in Section 5).
Fig. 4. Mutation Operators for GOAL
4 An Empirical Evaluation of Programs

We now turn to an empirical evaluation by examining a collection of GOAL programs. The aim of this examination is primarily to assess how well the mutation operators are able to generate realistic bugs. However, we also briefly consider what our empirical evaluation tells us about the two foundational hypotheses of mutation testing (Section 4.1).

Methodology: We obtained a collection of 55 GOAL programs, written as an assignment by first year undergraduate students at Delft university. These programs each implement a solution to “Blocks World for Teams” (BW4T) [7]: a single\(^3\) agent that moves around an environment with a number of rooms (see Figure 6), collecting blocks of various colours, and delivering them to the “dropzone” in a specified order (e.g. a red block, then a blue block). The environment (which runs in a separate process) provides the agent with percepts (e.g. in(Room), color(BlockID, Color), holding(BlockID)), and four actions (goTo(Location), goToBlock(BlockID), pickUp, and putDown).

We are interested in assessing how well the mutation operators are able to generate realistic bugs. We therefore consider the collection of GOAL programs as being a source of realistic buggy programs, and consider whether each of the buggy programs could have been generated from a correct program by applying our mutation operators. We therefore proceeded by testing each program to locate bugs, and then fixing the bugs. In fixing bugs we were careful to only make changes that were necessary, and to consider what alternative changes might be used to fix the bug. Once a bug was fixed we re-tested the program to confirm that the fix was correct. When testing the programs we used a

\(^3\) We only considered the initial version of the assignment with one agent.
number of test suites: the two example scenarios that were used in the original assignment, a set of ten randomly generated test cases, and a generated enumeration of all possible starting configurations within a limited scope (for scope sizes 1 and 2). Overall, we considered a program to be correct if it managed to deliver the desired blocks in all runs. Of the 55 programs, 4 were excluded, since they did not run at all (e.g. syntax errors), and 15 further programs were excluded since they did not have any (detected) bugs. This left a total of 36 buggy programs.

Before we consider how well the mutation operators are able to generate realistic bugs (as represented by the 36 buggy programs), we need to consider the assumptions that were made in developing the mutation operators. To what extent do these assumptions hold?

Recall that GOAL programs have a number of components (e.g. domain knowledge, action definitions), and that we have focussed on the program rules, i.e. assumed that errors only occur in the program rules. Is this a valid assumption? Out of the 36 programs with bugs, only 9 programs involved errors that related to non-supported GOAL features. One of these 9 programs involved incorrect usage of nested rules, and the remaining 8 programs had problems in the definition of actions, mostly incorrect definitions of their pre/post conditions. The somewhat surprising number of programs with issues in defining actions may be due to a feature of the BW4T environment: the environment is a separate process, and there is a delay between performing an action, and the action actually taking place. This means that when defining an action, such as pickUp, the action’s post-condition should be “true”, rather than, say, “holding(Block)”, because the environment will, in due course, perform the action and inform the agent of the action’s success (or failure) by sending suitable percepts (such as “holding(Block)”). Having pickUp make holding(Block) true is a problem because in reality (i.e. in the environment) the agent may fail to pick up the block, or may take a while to succeed. If holding(Block) is asserted immediately (by an incorrect post condition), then the agent may then proceed to move to the dropzone, based on the false belief that it has already picked up the block.

The second assumption that we made in developing the mutation operators was to ignore certain features specific to GOAL, namely modules, nested rules, and macros. This assumption was clearly reasonable: of the 36 buggy programs, only 6 programs used nested rules (2 of these 6 also used macros). However, only one of these 6 programs had a bug that was related to the use of nested rules.

Having considered, and evaluated, the assumptions, we now consider to what extent the bugs that we observed could be seen as the result of one or more applications of the defined mutation operators. As noted earlier, 27 out of the 36 buggy programs had bugs that solely related to supported GOAL features. Of these 27 programs, 16 programs had errors that did not require additional mutation operators. The remaining 11 programs had errors that corresponded to the application of a number of mutation operator instances, where at least one of the operators was additional to the ones that we had defined. The additional mutation operators were: (i) addition of elements (either actions or literals) [7 programs]; (ii) changes to the order of rules in a program other than the

4 Of these 8, one also had an error in the domain knowledge where a “&gt;” should have been “&ge;”, and two had incorrectly defined actions using e.g. pickUp(Block) instead of pickUp.
cases defined\(^5\) [3 programs]; (iii) mutating a variable to another (legal) variable name [3 programs]; and (iv) replacing an “insert” with a “drop” [1 program]. As discussed in Section 3, mutation operators that add to the program are problematic; however, the other three types of rules could be easily added.

Finally, we consider which of the mutation operators are used to generate the observed bugs. Figure 7 (right column) contains a summary of the number of times that each rule was used in deriving buggy programs, summed up over the 36 programs. Note that since some programs had bugs that corresponded to the application of multiple mutation operators, the sum of the number of rule applications (final row) is greater than the number of buggy programs. As can be seen in Figure 7, many of the rules that we defined actually do not correspond to the sorts of errors that we found in real buggy programs. Indeed, as often appears to be the case in mutation testing, only a few rules account for most of the bug types. For example, the four most commonly used rules (program:seqswap, program:drop1, LC:drop1, and AC:drop1, which are bolded in Figure 7) correspond to 71\% of the mutation operator applications.

Overall, we conclude that: 75\% (27 out of 36) of the programs had bugs that did not involve excluded GOAL features, such as modules, action definitions, or nested rules; 59\% of these (16 out of 27) had bugs that were able to be generated by the mutation operators that we defined; and many (71\%) of the mutation operator applications were instances of only four rules.

### 4.1 Evidence for the Foundational Hypotheses

Recall that the field of mutation testing rests on two foundational hypotheses. The *competent programmer hypothesis* states that programmers write programs that are “almost correct”, i.e. programs that are “a few mutants away from a correct program” \[10, \text{Page 531}\]. Of the 36 buggy programs, we found that 27 programs (75\%) were indeed a few mutants away from a correct program (defining “a few” to be “6 or fewer”). Thus we conclude that there is evidence that the competent programmer hypothesis holds for GOAL programs, even when they are written by first-year students. The table on the right shows how many programs were \(N\) mutation operators away from being correct. For instance, 7 of the 36 buggy programs corresponded to the application of a single mutation operator (first row), and 7 programs had 2 mutations (second row). The last row indicated that there were 4 programs that required more than 10 mutation operator applications: these required 11, 16, 20, and 22 applications respectively.

The *coupling effect hypothesis* states that a test set that is adequate with respect to single mutations is also adequate for multiple mutations. Since it concerns test sets and adequacy, this hypothesis is not easy to assess, and a full assessment is beyond the

\(^5\) These could, in principle, be regarded as repeated application of program:seqswap.
scope of this paper. However, we can provide some initial evidence: if, in fact, most of the observed bugs are generated by the application of a single mutation, then this would be evidence for the coupling effect hypothesis. Note that the converse is not true: even if the observed bugs mostly involve the application of multiple mutations, this does not mean that the coupling effect hypothesis fails to hold. There could be other mutants that can be used to assess whether the test suite is adequate with respect to the given bug. Considering the programs we found that of the 36 programs, 7 (19%) were generated by a single mutation operator application.

5 Implementation

The mutation operators defined in Figure 4 have been implemented. The implementation reads in a GOAL program and generates a collection of mutated programs, each of which is the result of applying a single mutation. The implementation considers mutations in both the main module, and in the percept processing module. It changes a single GOAL program rule, and then reassembles the complete program, including generating a modified mas2g file to run the mutated program.

We have run the implementation on three example GOAL programs (we selected the three longest examples in the GOAL distribution, excluding an example which uses modules extensively). The results were used in two ways. Firstly, we ran the mutants (for the 1st and 3rd programs) to check that each mutant was indeed a syntactically valid GOAL program (we couldn’t do this for the 2nd program, because it could not be run from the command line, due to the way the agent’s environment was implemented). Secondly, we observed which of the mutation operators were applicable to each program, and how many mutants were generated by the different rules. Figure 7 (middle columns) shows how many mutants were generated by the application of each of the mutation operators. Note that mutation operators that simply select part of a rule and invoke another mutation operator to make the change are not shown, since they do not actually change the program.

6 Discussion

We have presented rules for generating mutants of programs in a typical cognitive agent-oriented programming language (namely GOAL). We have also presented initial evidence that the rules are able to generate a significant proportion of the realistic bugs encountered in a simple problem, as well as evidence that supported the competent programmer foundational hypothesis of mutation testing in an agent context.

There are a number of issues (threats to validity) that need to be acknowledged. Firstly, we only considered a single program, and although we considered 55 different programs, all these programs only involved a single agent, and all were written by relatively inexperienced programmers. Clearly, one area for future work is to revisit the empirical evaluation using a wider range of problems, and a wider range of programmers. Note that these limitations are the reason why we have derived the mutation operators systematically based on the syntactical structure of GOAL, rather than by considering what mutation operators correspond to errors in the GOAL programs.
Another area for future work is assessing the prevalence of equivalent mutants, and whether equivalent mutants are generated by all mutation operators with roughly equal likelihood, or by certain rules.

We also intend to apply this approach to define mutation operators for other AO-PLs. Indeed, we have already defined mutation operators for AgentSpeak, but space precludes presenting them here.

More broadly, the data that we have collected also tells us information on the sorts of mistakes that (novice) GOAL programmers make. Analysing the data from this perspective would be valuable.

Acknowledgements: We would like to thank our colleagues at Delft university for providing access to the student projects, and for answering questions about GOAL, and fixing bugs that we found in the BW4T environment.

References

### Author Index

<table>
<thead>
<tr>
<th>Author</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abushark, Yoosef</td>
<td>49</td>
</tr>
<tr>
<td>Albayrak, Sahin</td>
<td>160</td>
</tr>
<tr>
<td>Baldoni, Matteo</td>
<td>17</td>
</tr>
<tr>
<td>Baroglio, Cristina</td>
<td>17</td>
</tr>
<tr>
<td>Bordini, Rafael H.</td>
<td>81</td>
</tr>
<tr>
<td>Burkhardt, Michael</td>
<td>160</td>
</tr>
<tr>
<td>Cao, Qiying</td>
<td>239</td>
</tr>
<tr>
<td>Capuzzimati, Federico</td>
<td>17</td>
</tr>
<tr>
<td>Cardoso, Rafael C.</td>
<td>81</td>
</tr>
<tr>
<td>Case, Denise</td>
<td>33</td>
</tr>
<tr>
<td>Colombetti, Marco</td>
<td>128</td>
</tr>
<tr>
<td>Cossentino, Massimo</td>
<td>65</td>
</tr>
<tr>
<td>Cranefield, Stephen</td>
<td>112</td>
</tr>
<tr>
<td>Dastani, Mehdi</td>
<td>270</td>
</tr>
<tr>
<td>Deloach, Scott</td>
<td>33</td>
</tr>
<tr>
<td>Dimuro, Graçaliz</td>
<td>97</td>
</tr>
<tr>
<td>Fornara, Nicoletta</td>
<td>128</td>
</tr>
<tr>
<td>Gerard, Scott</td>
<td>1</td>
</tr>
<tr>
<td>Ghose, Aditya</td>
<td>239</td>
</tr>
<tr>
<td>Gunay, Akin</td>
<td>192</td>
</tr>
<tr>
<td>Hessler, Axel</td>
<td>160</td>
</tr>
<tr>
<td>Hubner, Jomi F.</td>
<td>81</td>
</tr>
<tr>
<td>Jensen, Andreas Schmidt</td>
<td>224</td>
</tr>
<tr>
<td>Kaiser, Silvan</td>
<td>160</td>
</tr>
<tr>
<td>Kalia, Anup</td>
<td>1</td>
</tr>
<tr>
<td>Keiser, Jan</td>
<td>160</td>
</tr>
<tr>
<td>Konnerth, Thomas</td>
<td>160</td>
</tr>
<tr>
<td>Küster, Tobias</td>
<td>160</td>
</tr>
<tr>
<td>Liu, Wei</td>
<td>176</td>
</tr>
<tr>
<td>Lodato, Carmelo</td>
<td>65</td>
</tr>
<tr>
<td>Lopes, Salvatore</td>
<td>65</td>
</tr>
<tr>
<td>Luetzenberger, Marco</td>
<td>160</td>
</tr>
<tr>
<td>Luz, Bernardo</td>
<td>208</td>
</tr>
<tr>
<td>Masuch, Nils</td>
<td>160</td>
</tr>
<tr>
<td>Name</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Meneguzzi, Felipe</td>
<td>208</td>
</tr>
<tr>
<td>Molesini, Ambra</td>
<td>144</td>
</tr>
<tr>
<td>Morrison, Evan</td>
<td>239</td>
</tr>
<tr>
<td>Muscar, Alex</td>
<td>255</td>
</tr>
<tr>
<td>Okouya, Daniel</td>
<td>128</td>
</tr>
<tr>
<td>Omicini, Andrea</td>
<td>144</td>
</tr>
<tr>
<td>Padgham, Lin</td>
<td>176</td>
</tr>
<tr>
<td>Ranathunga, Surangika</td>
<td>112</td>
</tr>
<tr>
<td>Ribino, Patrizia</td>
<td>65</td>
</tr>
<tr>
<td>Rocha Costa, Antonio Carlos</td>
<td>97</td>
</tr>
<tr>
<td>Rodrigues, Thiago</td>
<td>97</td>
</tr>
<tr>
<td>Sabatucci, Luca</td>
<td>65</td>
</tr>
<tr>
<td>Savarimuthu, Bastin Tony Roy</td>
<td>239</td>
</tr>
<tr>
<td>Savarimuthu, Sharmila</td>
<td>286</td>
</tr>
<tr>
<td>Shi, Youqun</td>
<td>239</td>
</tr>
<tr>
<td>Singh, Munindar</td>
<td>1</td>
</tr>
<tr>
<td>Telang, Pankaj</td>
<td>1</td>
</tr>
<tr>
<td>Thangarajah, John</td>
<td>49, 176</td>
</tr>
<tr>
<td>Thiele, Alexander</td>
<td>160</td>
</tr>
<tr>
<td>Tonn, Jakob</td>
<td>160</td>
</tr>
<tr>
<td>van Zee, Marc</td>
<td>270</td>
</tr>
<tr>
<td>Vicari, Rosa</td>
<td>208</td>
</tr>
<tr>
<td>Viroli, Mirko</td>
<td>144</td>
</tr>
<tr>
<td>Winikoff, Michael</td>
<td>286</td>
</tr>
<tr>
<td>Wong, Wilson</td>
<td>176</td>
</tr>
<tr>
<td>Xu, Hongyun</td>
<td>239</td>
</tr>
<tr>
<td>Yolum, Pinar</td>
<td>192</td>
</tr>
<tr>
<td>Zambonelli, Franco</td>
<td>144</td>
</tr>
</tbody>
</table>