Mehdi Dastani
Amal El Fallah Seghrouchni
Jomi Hübner
João Leite (Eds.)

LNAI 6822

Languages, Methodologies, and Development Tools for Multi-Agent Systems

Third International Workshop, LADS 2010
Lyon, France, August/September 2010
Revised Selected Papers

Springer

Mehdi Dastani   Amal El Fallah Seghrouchni
Jomi Hübner   João Leite (Eds.)

Languages, Methodologies, and Development Tools for Multi-Agent Systems

Third International Workshop, LADS 2010
Lyon, France, August 30 – September 1, 2010
Revised Selected Papers

Springer

Series Editors

Randy Goebel, University of Alberta, Edmonton, Canada
Jörg Siekmann, University of Saarland, Saarbrücken, Germany
Wolfgang Wahlster, DFKI and University of Saarland, Saarbrücken, Germany

Volume Editors

Mehdi Dastani
Utrecht University, Intelligent Systems Group
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: mehdi@cs.uu.nl

Amal El Fallah Seghrouchni
LIP6 – University Pierre and Marie Curie
104 Avenue du Président Kennedy, 75016 Paris, France
E-mail: amal.elfallah@lip6.fr

Jomi Hübner
Federal University of Santa Catarina, Dept. of Automation and Systems Engineering
P.O.Box 476, Florianópolis, SC 88040-900, Brazil
E-mail: jomi@das.ufsc.br

João Leite
CENTRIA, Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal
E-mail: jleite@di.fct.unl.pt

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-22722-6 e-ISBN 978-3-642-22723-3
DOI 10.1007/978-3-642-22723-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011935119

CR Subject Classification (1998): I.2.11, I.2, I.6, H.3.5, D.2

LNCS Sublibrary: SL 7 – Artificial Intelligence

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This book contains the proceedings of the Third International Workshop on Languages, Methodologies and Development Tools for Multi-agent Systems (LADS 2010), which took place at the Domaine Valpré in Lyon, France, from August 30 to September 1, 2010. As in the previous two editions, this workshop was a part of MALLOW, a federation of workshops on Multi-Agent Logics, Languages, and Organizations.

The LADS 2010 workshop addressed both theoretical and practical issues related to developing and deploying multi-agent systems. It constituted a rich forum where leading researchers from both academia and industry could share their experiences on formal approaches, programming languages, methodologies, tools and techniques supporting the development and deployment of multi-agent systems. From a theoretical point of view, LADS 2010 aimed at addressing issues related to theories, methodologies, models and approaches that are needed to facilitate the development of multi-agent systems ensuring their predictability and verification. Formal declarative models and approaches have the potential of offering solutions for the specification and design of multi-agent systems. From a practical point of view, LADS 2010 aimed at stimulating research and discussion on how multi-agent system specifications and designs can be effectively implemented and tested.

This book is the result of a strict selection and review process. From 11 papers originally submitted to LADS 2010, together with 2 invited submissions, after 2 rounds of reviews we selected 8 high-quality papers covering important topics related to multi-agent programming technology.

We would like to thank all authors, Programme Committee members, and additional reviewers for their outstanding contribution to the success of LADS 2010. We would also like to thank all the sponsors. We are particularly grateful to Olivier Boissier, Salima Hassas, Nicolas Maudet and all the MALLOW Organizing Committee for their technical support and for hosting LADS 2010.

May 2011

Mehdi Dastani
Amal El Fallah Seghrouchni
Jomi Hübner
João Leite

Organization

Workshop Chairs

Mehdi Dastani	Utrecht University, The Netherlands
Amal El Fallah Seghrouchni	University of Paris VI, France
Jomi Hübner	Federal University of Santa Catarina, Brazil
João Leite	Universidade Nova de Lisboa, Lisbon, Portugal

Programme Committee

Marco Alberti	Universidade Nova de Lisboa, Portugal
José Júlio Alferes	Universidade Nova de Lisboa, Portugal
Matteo Baldoni	Univesità degli Studi di Torino, Italy
Juan Botia	Universidad de Murcia, Spain
Lars Braubach	University of Hamburg, Germany
Yves Demazeau	CNRS - Laboratoire LIG, France
Juergen Dix	Clausthal University of Technology, Germany
Paolo Giorgini	University of Trento, Italy
Koen Hindriks	Delft University of Technology, The Netherlands
Shinichi Honiden	National Institute of Informatics, Japan
Wojtek Jamroga	University of Luxembourg
Peep Küngas	University of Tartu, Estonia
Brian Logan	University of Nottingham, UK
Alessio Lomuscio	Imperial College London, UK
Viviana Mascardi	Università degli Studi di Genova, Italy
John-Jules Meyer	Utrecht University, The Netherlands
Alexander Pokahr	University of Hamburg, Germany
Alessandro Ricci	University of Bologna, Italy
Patrick Taillibert	Thales Aerospace Division, France
Paolo Torroni	University of Bologna, Italy
Leon van der Torre	University of Luxembourg
M. Birna van Riemsdijk	Delft University of Technology, The Netherlands
Pinar Yolum	Bogazici University, Turkey
Yingqian Zhang	Erasmus University Rotterdam, The Netherlands

Additional Reviewers

Natasha Alechina
Cristina Baroglio
Tristan Behrens
Francesco Belardinelli
Akin Gunay
Ozgur Kafali
Elisa Marengo
Yasuyuki Tahara

LADS Steering Committee

Mehdi Dastani	Utrecht University, The Netherlands
Amal El Fallah Seghrouchni	University of Paris VI, France
João Leite	Universidade Nova de Lisboa, Portugal
Paolo Torroni	University of Bologna, Italy

Table of Contents

OperettA: Organization-Oriented Development Environment*

Huib Aldewereld and Virginia Dignum

[1] Utrecht University, The Netherlands
huib@cs.uu.nl
[2] Delft University of Technology, The Netherlands
m.v.dignum@tudelft.nl

Abstract. The increasing complexity of distributed applications requires new modeling and engineering approaches. Such domains require representing the regulating structures explicitly and independently from the acting components (or agents). Organization computational models, based on Organization Theory, have been advocated to specify such systems. In this paper, we present the organizational modeling approach OperA and a graphical environment for the specification and analysis of organizational models, OperettA. OperA provides an expressive way for defining open organizations distinguishing explicitly between the organizational aims, and the agents who act in it. That is, OperA enables the specification of organizational structures, requirements and objectives, and at the same time allows participants to have the freedom to act according to their own capabilities and demands. OperettA takes a Model Driven Engineering approach combining different formal methods and enables model validation.

1 Introduction

The engineering of applications for complex and dynamic domains is an increasingly difficult process. Requirements and functionalities are not fixed a priori, components are not designed nor controlled by a common entity, and unplanned and unspecified changes may occur during runtime. There is a need for representing the regulating structures explicitly and independently from the acting components (or agents). Organization computational models, based on Organization Theory, have been advocated to specify such systems. Comprehensive analysis of several agent systems has shown that different design approaches are appropriate for different domain characteristics [4]. In particular, agent organization frameworks are suitable to model complex environments where many independent entities coexist witing explicit normative and organizational structures and global specification of control measures is necessary.

* This work has been performed in the framework of the FP7 project ALIVE IST-215890, which is funded by the European Community. The authors would like to acknowledge the contributions of their colleagues from ALIVE Consortium (http://www.ist-alive.eu).

A reason for creating organizations is efficiency; that is, to provide the means for coordination that enables the achievement of global goals in an efficient manner. This indicates distribution of functions and coordination of activity such that 'the right agent is doing the right thing' [3].

Organization models comprise structural and behavioral aspects [16,11]. Structural aspects include the formal patterns of relationships between groups and individuals, and the norms that govern their interactions, while behavioral aspects include processes, rules, activities, and operational methods.

Comprehensive models for organizations must, on the one hand, be able to specify global goals and requirements but, on the other hand, cannot assume that participating actors will always act according to the needs and expectations of the system design. Concepts as organizational rules [23], norms and institutions [7], [8], and social structures [15] arise from the idea that the effective engineering of organizations needs high-level, actor-independent concepts and abstractions that explicitly define the organization in which agents live [24]. These are the rules and global objectives that govern the activity of an organization.

In this paper, we present a graphical environment for the specification and analysis of organizational models, based on the OperA formalism [2]. This organization specification tool builds heavily on mechanisms from Model Driven Engineering (MDE), which enables the introduction and combination of different formal methods hence enabling the modeling activity through systematic advices and model design consistency checking.

The paper is organized as follows: first, in section 2 we introduce and briefly explain the OperA framework. In section 3 the specification of organizational models in OperA is detailed. In section 4 we introduce the OperettA Environment, which is an MDE-based graphical editor for the specification and validation of OperA organizational models. Finally, section 6 gives some conclusions.

2 Organization Modeling: The OperA Framework

Organization models, combining global requirements and individual initiative, have been advocated to specify open systems in dynamic environments [9,2]. We take formal processes and requirements as a basis for the modeling of complex systems that regulate the action of the different agents. Organizational models must enable the explicit representation of structural and strategic concerns and their adaptation to environment changes in a way that is independent from the behavior of the agents.

The deployment of organizations in dynamic and unpredictable settings brings forth critical issues concerning the design, implementation and validation of their behavior [16,11,22], and should be guided by two principles.

- Provide sufficient representation of the institutional requirements so that the overall system complies with the norms.
- Provide enough flexibility to accommodate heterogeneous components.

Therefore, organizational models must provide means to represent concepts and relationships in the domain that are rich enough to *cover* the necessary contexts of agent interaction while keeping in mind the *relevance* of those concepts for the global aims of the system.

The OperA model [2] proposes an expressive way for defining open organizations distinguishing explicitly between the organizational aims, and the agents who act in it. That is, OperA enables the specification of organizational structures, requirements and objectives, and at the same time allows participants to have the freedom to act according to their own capabilities and demands. At an abstract level, an OperA model describes the aims and concerns of the organization with respect to the social system. These are described as organization's externally observable *objectives*, that is, the desired states of affairs for the organization.

The OperA framework consists of three interrelated models. The **Organizational Model** (OM) is the result of the observation and analysis of the domain and describes the desired behavior of the organization, as determined by the organizational stakeholders in terms of objectives, norms, roles, interactions and ontologies. The OM will be described in more detail in section 3, using as example the conference organization scenario taken from [6].

The OM provides the overall organization design that fulfills the stakeholders requirements. Objectives of an organization are achieved through the action of agents, which means that, at each moment, an organization should employ the relevant agents that can make its objectives happen. However, the OM does not specify how to structure groups of agents and constrain their behavior by social rules such that their combined activity will lead to the desired results. The **Social Model** (SM) maps organizational roles to agents and describes agreements concerning the role enactment and other conditions in social contracts. Finally, the **Interaction Model** (IM) specifies the interaction agreements between role-enacting agents as interaction contracts. IM specification enable variations to the enactment of interactions between role-enacting agents. In section 4.2 we describe the use of MDE principles to implement this framework.

3 The Organization Model

A common way to express the objectives of an organization is in terms of its expected functionality, that is, what is the organization expected to do or produce.In OperA, the Organization Model (OM) specifies the *means* to achieve such objectives. That is, OM describes the structure and global characteristics of a domain from an organizational perspective, where global goals determine roles and interactions, specified in terms of its *Social* and *Interaction Structures*. E.g., how should a conference be organized, its program, submissions, etc.

Moreover, organization specification should include the description of concepts holding in the domain, and of expected or required behaviors.Therefore, these structures should be linked with the norms, defined in *Normative Structure*, and with the ontologies and communication languages defined in the *Communication Structure*.

3.1 The Social Structure

The social structure of an organization describes the roles holding in the organization. It consists of a list of role definitions, *Roles* (including their objectives, rights and requirements), such as PC-member, program chair, author, etc.; a list of role groups' definitions, *Groups*; and a *Role Dependencies* graph.

Abstract society objectives form the basis for the definition of the objectives of roles. *Roles* are the main element of the *Social Structure*. From the society perspective, role descriptions should identify the activities and services necessary to achieve society objectives and enable to abstract from the individuals that will eventually perform the role. From the agent perspective, roles specify the expectations of the society with respect to the agent's activity in the society. In OperA, the definition of a role consists of an identifier, a set of role objectives, possibly sets of sub-objectives per objective, a set of role rights, a set of norms and the type of role. An example of role description is presented in table 1.

Table 1. *PC member* role description

Id	PC_member
Objectives	paper_reviewed(Paper,Report)
Sub-objectives	{read(P), report_written(P, Rep), review_received(Org, P, Rep)}
Rights	access-confmanager-program(*me*)
Norms & Rules	PC_member is OBLIGED to understand English IF paper_assigned THEN PC_member is OBLIGED to review paper BEFORE given deadline IF author of paper_assigned is colleague THEN PC_member is OBLIGED to refuse to review asap

Groups provide means to collectively refer to a set of roles and are used to specify norms that hold for all roles in the group. Groups are defined by means of an identifier, a non-empty set of roles, and group norms. An example of a group in the conference scenario is the organizing team consisting of the roles *program chair*, *local organizer*, and *general chair*.

The distribution of objectives in roles is defined by means of the *Role Hierarchy*. Different criteria can guide the definition of *Role Hierarchy*. In particular, a role can be refined by decomposing it in sub-roles that, together, fulfill the objectives of the given role.

This refinement of roles defines *Role Dependencies*. A dependency graph represents the dependency relations between roles. Nodes in the graph are roles in the society. Arcs are labelled with the objectives for which the parent role depends on the child role. Part of the dependency graph for the conference society is displayed in figure 1. For example, the arc between nodes PC-Chair and PC-member represents the dependency between *PC-Chair* and *PC-member* concerning *paper-reviewed* ($PC - Chair \succeq_{paper\_reviewed} PC - Member$). The way

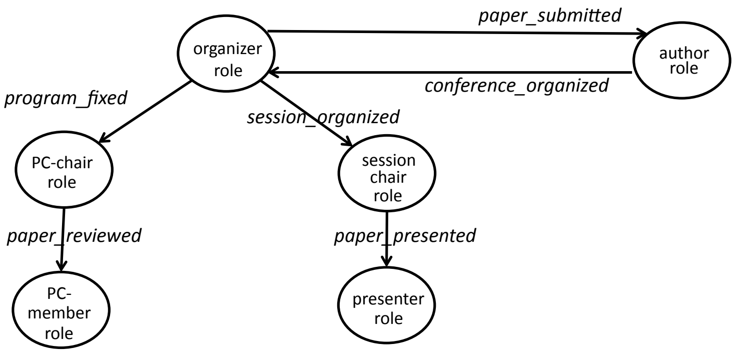


Fig. 1. Role dependencies in a conference

objective g in a dependency relation $r_1 \succeq_g r_2$ is actually passed between $r1$ and $r2$ depends on the coordination type of the society, defined in the Architectural Templates. In OperA, three types of role dependencies are identified: *bidding*, *request* and *delegation*.

3.2 The Interaction Structure

Interaction is structured in a set of meaningful scenes that follow pre-defined abstract scene scripts. Examples of scenes are the registration of participants in a conference, which involves a representative of the organization and a potential participant, or paper review, involving program committee members and the PC chair. A *scene script* describes a scene by its players (roles), its desired results and the norms regulating the interaction. In the OM, scene scripts are specified according to the requirements of the society. The results of an interaction scene are achieved by the joint activity of the participating roles, through the realization of (sub-)objectives of those roles. A scene script establishes also the desired *interaction patterns* between roles, that is, a desired combination of the (sub-) objectives of the roles. Table 2 gives an example of a scene script.

OperA interaction descriptions are declarative, indicating the global aims of the interaction rather than describing exact activities in details. Interaction objectives can be more or less restrictive, giving the agent enacting the role more or less freedom to decide how to achieve the role objectives and interpret its

Table 2. Script for the *Review Process* scene

Scene	Review Process
Roles	Program-Chair (1), PC-member(2..Max)
Results	$r_1 = \forall$ P $\in$ Papers: reviews_done(P, rev1, rev2)
Interact. Pattern	PATTERN(r_1): *see figure 2*
Norms & Rules	Program-Chair is PERMITTED to assign papers PC-member is OBLIGED to review papers assigned before deadline

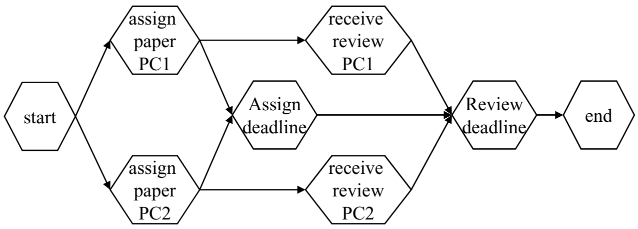


Fig. 2. Landmark pattern for *Review Process*

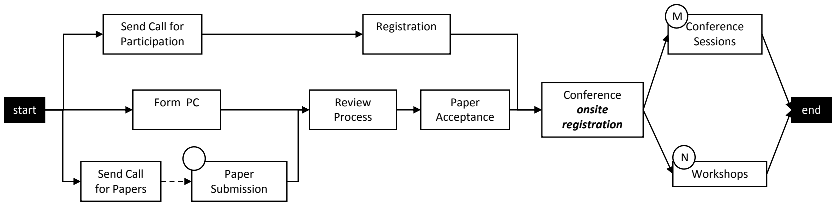


Fig. 3. Interaction Structure in the conference scenario

norms. Following the ideas of [17,13], we call such expressions *landmarks*, defined as conjunctions of logical expressions that are true in a state. Landmarks combined with a partial ordering to indicate the order in which the landmarks are to be achieved are called a *landmark pattern*. Figure 2 shows the landmark pattern for the *Review Process*. Several different specific actions can bring about the same state, that is, landmark patterns actually represent families of protocols. The use of landmarks to describe activity enables the actors to choose the best applicable actions, according to their own goals and capabilities. The relation between scenes is represented by the *Interaction Structure* (see figure 3). In this diagram, *transitions* describe a partial ordering of the scenes, plus eventual synchronization constraints. Note that several scenes can be happening at the same time and one agent can participate in different scenes simultaneously. Transitions also describe the conditions for the creation of a new instance of the scene, and specify the maximum number of scene instances that are allowed simultaneously. Furthermore, the enactment of a role in a scene may have consequences in following scenes. Role *evolution relations* describe the constraints that hold for the role-enacting agents as they move from scene to scene.

3.3 The Normative Structure

At the highest level of abstraction, norms are the *values* of a society, in the sense that they define the concepts that are used to determine the value or utility of situations. For the conference organization scenario, the desire to share information and uphold scientific quality can be seen as values. However, values do not specify *how*, *when* or in *which* conditions individuals should behave appropriately in any given social setup. In OperA, these aspects are defined in the Normative Structure. For example, the *information sharing* value can be described as $D(share(info))$. However, besides a formal syntax, this does not provide any meaning to the concept of *value*.

In OperA, norms are specified using a deontic logic that is temporal, relativized (in terms of roles and groups) and conditional. For instance, the following norm might hold: *"The authors should submit their contributions before the deadline"*, which can be formalized as: $O_{author}(submit(paper) \leq Deadline)$.

Furthermore, in order to check norms and act on possible violations of the norms by the agents within an organization, abstract norms have to be translated into actions and concepts that can be handled within such organizations. To do so, the definition of the abstract norms are iteratively concretized into more concrete norms, and then translated into specific rules, violations and sanctions.

Concrete norms are related to abstract norms through a mapping function, based on the counts-as operator as developed in [1]. For example, in the context of Org, $submit(paper)$ can be concretized as:$send\_mail(organizer, files) \vee send\_post(organizer, hard\_copies) \rightarrow_{Org} submit(paper)$.

3.4 The Communication Structure

Communication mechanisms include both the representation of domain knowledge (*what* are we talking about) and protocols for communication (*how* are we talking). Both content and protocol have different meanings at the different levels of abstraction (e.g. while at the abstract level one might talk of *disseminate*, such action will most probably not be available to agents acting at the implementation level). Specification of communication content is usually realized using ontologies, which are shared conceptualizations of the terms and predicates in a domain. Agent communication languages (ACLs) are the usual means in MAS to describe communicative actions. ACLs are wrapper languages in the sense that they abstract from the content of communication.

In OperA, the Communication Structure describes both the content and the language for communication. The content aspects of communication, or domain knowledge, are specified by *Domain Ontologies* and *Communication Acts* define the language for communication, including the performatives and the protocols.

4 OperettA Environment

In order to support developers designing and maintaining organization models, tools are needed that provide an organization-oriented development environment. The requirements for such a development environment are the following.

1. Organizational Design: The tool should provide means for designing organizational models in an 'intuitive' manner. The tool should allow users to create and represent organizational structures, define the parties involved in an organization, represent organizational and role objectives, and define the pattern of interactions typically used to reach these objectives.

2. Organizational Verification: The tool should provide verification means and assistance in detecting faults in organizational designs as early as possible, to prevent context design issues from being translated to the other levels of system specification.
3. Ontology Design: The tool should to be able to specify, import, and maintain domain ontologies. Domain ontologies specifying the knowledge for a specific domain of interaction should be able to be represented, existing ontologies containing such information should be able to be included (and provide inputs for organizational concepts, such as role or objective names). Included or earlier created ontologies should be maintainable and updatable.
4. Connectivity to System Level: The output of the organizational design tool is intended for use by system level tools, namely MAS environments and agent programming languages. The output of the tool thus needs to provide easy integration and connection between the organization and system level.
5. User-Friendly GUI: A user-friendly graphical interface is to be provided for users to create and maintain organizational models easily. Help and guidelines are useful for beginners to use the tool.
6. Availability: The tool should be available under open source license and for use by other projects.

We have developed the OperettA development environment as an open-source solution on the basis of these requirements. OperettA enables the specification of OperA OMs, which satisfies the first requirement. OperettA combines multiple editors into a single package. It provides separate editors on different components of organizational models; i.e., it has different (graphical) editors for each of the main components of an organizational model as defined in the OperA framework. These specialized editors correspond to the OperA OM structures: social, interaction, normative and communicative. OperettA (as well as additional documentation, examples and tutorials) can be downloaded from `http://www.cs.uu.nl/research/projects/opera/`.

The OperettA tool is a combination of tools based on the Eclipse Modeling Framework (EMF) [18] and tools based on the Graphical Modeling Framework (GMF) integrated into a single editor. Developed as an Eclipse plug-in, OperettA is fully open-source and follows the MDE principles of tool development. In the following we look at the editors provided by OperettA, and how OperettA connects to MAS solutions, thus satisfying requirement 4.

4.1 OperettA Components

The main element of OperettA (see figure 4 for an overview of the tools in OperettA and their provided functionalities) is the OperA Meta-Model. The meta-model, created with the EMF tools, provides the (structural) definition of what organizational models should look like. This meta-model is extended with the default EMF edit and editor plug-ins to provide model accessors and the basic tree-based editor for the creation and management of OperA models. This basic editor has been extended with graphical interfaces for editing parts of

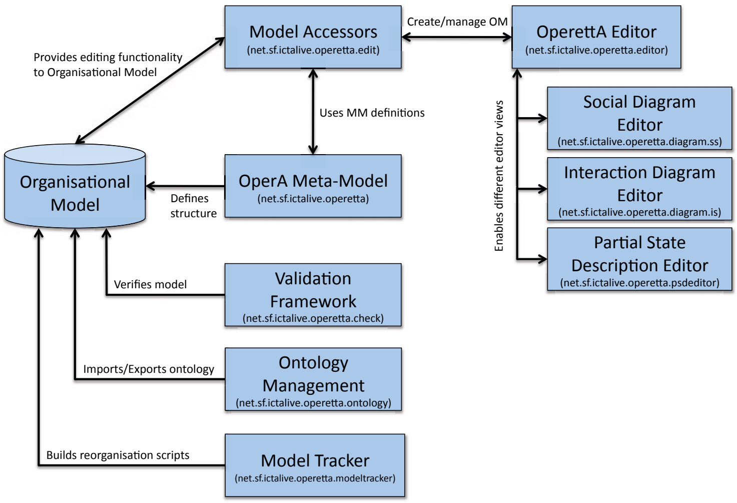

Model Accessors
(net.sf.ictalive.operetta.edit)
Create/manage OM
OperettA Editor
(net.sf.ictalive.operetta.editor)
Provides editing functionality to Organisational Model
Uses MM definitions
Enables different editor views
Social Diagram Editor
(net.sf.ictalive.operetta.diagram.ss)
Interaction Diagram Editor
(net.sf.ictalive.operetta.diagram.is)
Partial State Description Editor
(net.sf.ictalive.operetta.psdeditor)
Organisational Model
OperA Meta-Model
(net.sf.ictalive.operetta)
Defines structure
Verifies model
Validation Framework
(net.sf.ictalive.operetta.check)
Imports/Exports ontology
Ontology Management
(net.sf.ictalive.operetta.ontology)
Builds reorganisation scripts
Model Tracker
(net.sf.ictalive.operetta.modeltracker)

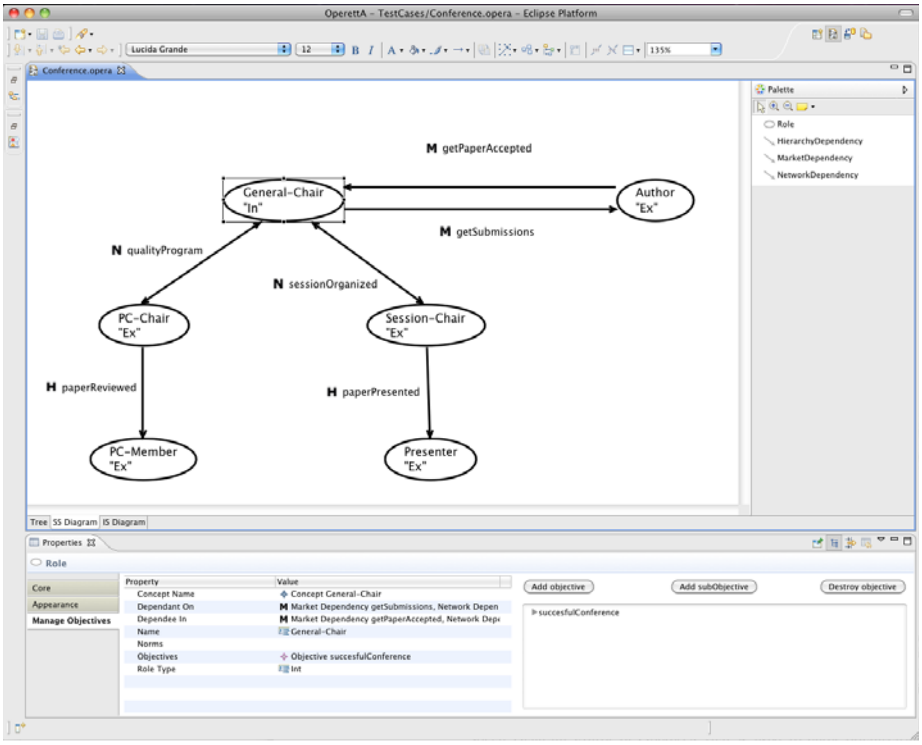
Fig. 4. OperettA tool components

the organization model: the social diagram editor, and the interaction diagram editor. A third extention is provided in the partial state description editor which allows for the inputting and editing of formulas (e.g., as part of the specification of a norm).

Next to the (graphical) editing extensions, OperettA contains three other plug-ins for additional functionality. The Validation Framework provides an improvement over the default validation of EMF-based tools to provide validation of additional restrictions. An ontology managing plug-in is included as well to allow the ontology developed with OperettA to be exported to OWL, as well as allowing for importing existing ontology into the organization to boot-strap the organization design. Finally, OperettA contains a Model Tracker plug-in that can generate re-organization descriptions based on changes made in the OperettA organization editors.

We discuss the graphical editors and additional plug-ins in more details in the following.

Social Diagram Editor. This graphical editor provides a view of the Social Structure element of OMs. It allows the graphical creation of organizational Roles and Dependencies, thus specifying the social relations between important parties that play a part in the organization. The Social Diagram Editor also provides editing capabilities to specify and manage Role related Objectives, to provide context for the different Roles in an organization. Figure 5 depicts the Social Diagram Editor of OperettA that is used to enter organizational roles and dependencies between roles. Role objectives are created and managed via the objectives editor shown in the bottom part of the figure.

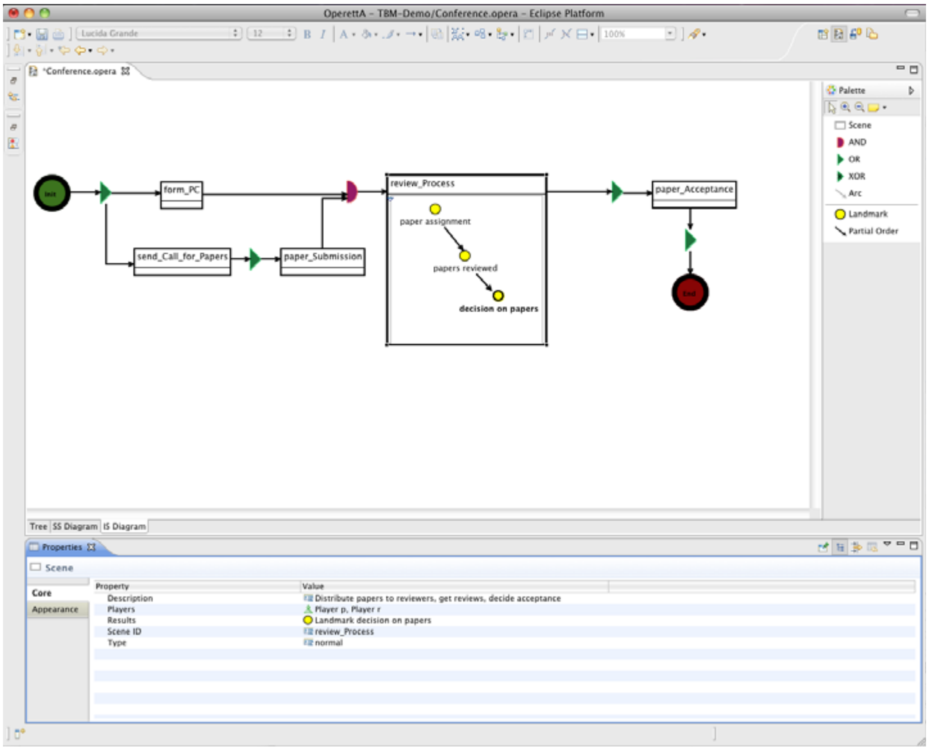


Property	Value
Concept Name	Concept General-Chair
Dependant On	M Market Dependency getSubmissions, Network Depen
Dependee In	M Market Dependency getPaperAccepted, Network Dep
Name	General-Chair
Norms	
Objectives	Objective succesfulConference
Role Type	Int

Fig. 5. OperettA social diagram editor

Interaction Diagram Editor. Similar to the Social Diagram Editor, the Interaction Diagram Editor provides a graphical view of the Interaction Structure element of OMs. This editor allows for the specification and management of the interaction elements of the organization; that is, it is for the specification and management of the different interactions that take place in the organization in order to achieve the different (role) objectives specified in the social part of the OM. The specification of the interaction is done in terms of scenes and transitions (the connection and synchronization points between scenes). Together, these define the order in which objectives are to be reached and how the organization works (though specified on a high level of abstraction). Moreover, the Interaction Diagram Editor allows for the specification and management of the Landmark Pattern within scenes. The Landmark Pattern of a scene describes how the scene is to be played, on the basis of the important states of the organization (called landmarks) and a specification of the order in which these are to be achieved (thus, defining a partial ordering over the landmarks). Figure 6 gives an overview of the Interaction Diagram Editor, with one of the scenes "opened" to show its landmark pattern.

Ontology Manager. The Ontology Manager part of the OperettA tool is a plug-in for importing and exporting (domain) ontologies. The creation and



Property	Value
Description	Distribute papers to reviewers, get reviews, decide acceptance
Players	Player p, Player r
Results	Landmark decision on papers
Scene ID	review_Process
Type	normal

Fig. 6. OperettA interaction diagram editor

maintenance of ontologies is done by external tools (like, for example, Protégé). Parts of the functionality of organizational ontology editing is included in the OperettA editors:

– Automatic creation of organizational ontology while designing the organization. As the designer is inputting the organizational model in OperettA, OperettA maintains an ontology of role names, objective names, and logical atoms that the designer uses to define the organization.
– Using an included (existing) ontology for the naming of organizational model elements; that is, if an (external) ontology is present in the organizational model it can be used to pick concept names for different parts of an organizational model (e.g., the name of a role can be picked from an existing ontology included in the model). The addition of the (external) ontology to an organizational model is done via the ontology manager.

The functionality of ontology editing in the OperettA tools is limited to organizational ontologies. The Ontology manager plug-in extends OperettA with the following capabilities:

– Importing an ontology from a file (e.g., rdf or owl [21]). Ontologies about the domain or organization that is to provide the context of a system might be

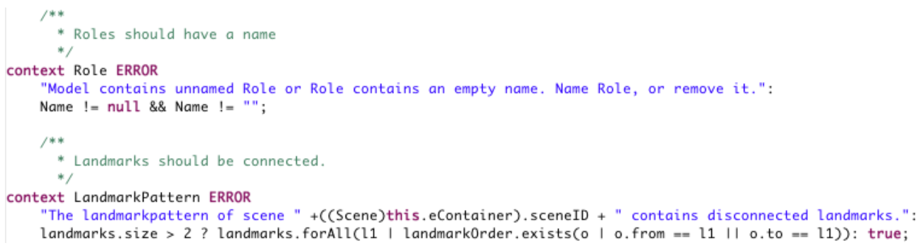
  already available. These ontologies tend to be stored in some conventional ontology file-format. The Ontology Manager allows OperettA to import and use such ontologies.
- Exporting (generated) organizational ontologies to file. In order to align an use the organizational ontology created by OperettA, the Ontology Manager extends the OperettA tool with the capability to export the default ontology (the ontology that is automatically created by OperettA) to an owl file.

The organizational ontology created by OperettA is stored in the Organizational Model. The ontological elements need to be available to the system level of design, and thus need to be included in the domain ontology. The integration of organizational concepts in a domain ontology is not trivial, as it should respect the structure of the domain ontology while adding organizational concepts as roles, objectives, etc. and the instances of these concepts; role names, objective names, etc. The alignment between the exported ontology and the domain ontology will have to be done by hand in an external editor, which is a discussion out of the scope of this paper. The inclusion of the ontology manager satisfies requirement 3.

Model Tracker. To support reorganization, OperettA is extended with a model tracker. This model tracker allows a designer to view the changes made on the organizational model since a last save (but not necessarily the previous one). By storing the changes to the organizational model in a history file, the model tracker can be used to generate scripts that express how an organization is changed. Reorganization scripts capture changes in a precise and concise manner, and can be used to communicate organizational changes to the system level.

Validation Framework. The validation plug-in of OperettA overwrites the basic validation provided by the EMF framework. Instead of just verifying constraints specified in the OperA meta-model, the validation has been extended with additional verification constraints to minimize organizational design mistakes. The overall purpose of the validation plug-in is to provide OM designers meaningful feedback to eliminate design errors as early as possible (in the design process). The validation plug-in is installed separately from OperettA, but after installation it can be invoked from within each of the different OperettA editing views. The validation plug-in seamlessly overwrites the standard EMF validation, making it the new default manner of validating OperettA models.

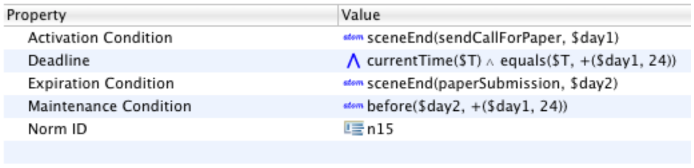
The validation plug-in works directly on the model instance to verify various modeling constraints, accessing the model via the meta-model definitions. Some examples of the constraints validated are checking that roles have a name, checking that role names are unique, checking that all roles have an objective, and so on. Less stringent constraints are checked as well, like, for example, whether roles are connected to other roles via dependencies; i.e., while it does not hold for every OM, in most models roles should be connected to other roles (that is, it should be depending upon (an)other role(s) or being depended upon by (an)other role(s)). Such "soft" constraints are presented to designer as a



```
    /**
      * Roles should have a name
      */
context Role ERROR
    "Model contains unnamed Role or Role contains an empty name. Name Role, or remove it.":
    Name != null && Name != "";

    /**
      * Landmarks should be connected.
      */
context LandmarkPattern ERROR
    "The landmarkpattern of scene " +((Scene)this.eContainer).sceneID + " contains disconnected landmarks.":
    landmarks.size > 2 ? landmarks.forAll(l1 | landmarkOrder.exists(o | o.from == l1 || o.to == l1)): true;
```

Fig. 7. Example validation constraints



Property	Value
Activation Condition	sceneEnd(sendCallForPaper, $day1)
Deadline	currentTime($T) ∧ equals($T, +($day1, 24))
Expiration Condition	sceneEnd(paperSubmission, $day2)
Maintenance Condition	before($day2, +($day1, 24))
Norm ID	n15

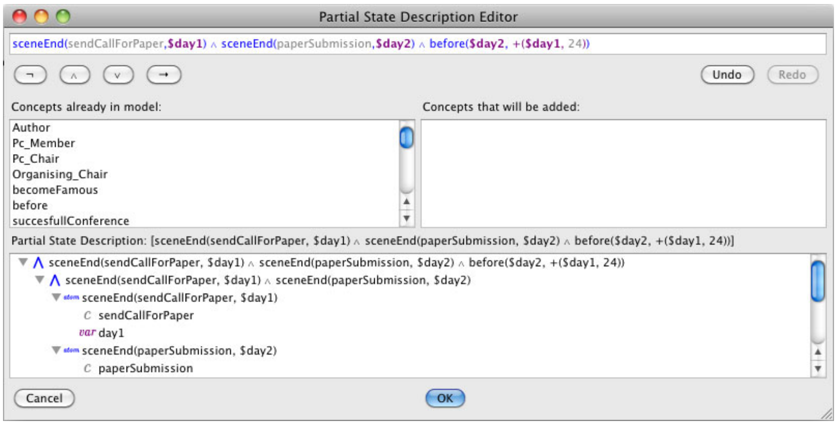
Fig. 8. A norm in OperettA

warning, intended to have the designer rethink their model and update if appropriate. The validation plugin fulfills requirement 2.

The validation framework uses OCL constraint language to specify the checks to be made on the OperA model. Figure 7 shows two example constraints from the validation framework. The first constraint is similar to the kind of constraints checked by the EMF framework, that is, checking whether the model is correctly filled according to the meta-model specifications. In the case shown, it is checked that Roles have a Name.

The other constraint shown in figure 7 is checking a methodological constraint, that is, whether the model is specified correctly according to OperA requirements. In the case shown below, it is checked that the landmarks in a landmark pattern are connected, that is, that a landmark pattern does not contain disconnected parts. This sort of constraints cannot be expressed in EMF, but can be expressed and validated with the validation framework.

Partial State Description Editor. Formulas are an important part of the organizational model, as they provide the necessary semantics of the elements. Formulas are located in several places, such as in the specification of *Objectives*, *Landmarks* and *Norms*. The norms are particularly important as they provide guidelines on a high level of abstraction for the participants to follow. An example norm in OperettA is shown in figure 8. Norms in OperettA are expressed in a formalization similar to [14] The norm shown in this figure describes that the paper submission scene should end within 24 days after the call for papers scene has ended. The norm is input using several logical formulas; one for the activation condition expressing when the norm is active (the send call for paper scene has ended), one for the expiration condition expressing when the norm is no longer



```
sceneEnd(sendCallForPaper,$day1) ∧ sceneEnd(paperSubmission,$day2) ∧ before($day2, +($day1, 24))

Concepts already in model:
Author
Pc_Member
Pc_Chair
Organising_Chair
becomeFamous
before
succesfullConference

Concepts that will be added:

Partial State Description: [sceneEnd(sendCallForPaper, $day1) ∧ sceneEnd(paperSubmission, $day2) ∧ before($day2, +($day1, 24))]

▼ ∧ sceneEnd(sendCallForPaper, $day1) ∧ sceneEnd(paperSubmission, $day2) ∧ before($day2, +($day1, 24))
  ▼ ∧ sceneEnd(sendCallForPaper, $day1) ∧ sceneEnd(paperSubmission, $day2)
    ▼ atom sceneEnd(sendCallForPaper, $day1)
        C sendCallForPaper
        var day1
    ▼ atom sceneEnd(paperSubmission, $day2)
        C paperSubmission
```

Fig. 9. OperettA partial state description editor

active (the paper submission scene has ended), one for the maintenance condition expressing the formula to be checked when the norm is active to see if violations have happened (the paper submission happens within 24 days of the call for papers), and one for the deadline expressing an explicit state of affairs when the norm should have been acted upon (the norm conditions are supposed to be all filled, except for the deadline, which may be omitted).

Inputting the various logical formulas contained in a norm (or in the objectives or landmarks) is done in a straightforward manner via the Partial State Description Editor. The Partial State Description Editor allows users to enter the formula by either typing or clicking. The editor automatically parses the formula being input and colors the elements to highlight the various elements and provide immediate feedback on whether the norm is parsed as intended by the user. Figure 9 shows the Partial State Description Editor in operation.

The top field of the editor allows the user to input the formula by typing (hovering over the operator buttons gives a hint about the commands that can be used to type the operators). The middle two lists give an overview of the concepts available in the model (left list) and the concepts that will be added when the formula is added (right list). The bottom view provides a preview of the formula presented in its EMF format, which can be used to check whether the elements are parsed in the correct manner.

This extension, with the graphical editors for the social and interaction structures, fulfills requirement 5.

4.2 Connectivity to System Level

The OperettA tool has only off-line functionalities; it is used by designers to create the context of the system and their linked ontologies. It provides design and validation functionalities for the creation and management of OperA organizational models. The models generated by the OperettA tool are XML specifications that comply with the OperA meta-model, a fragment of this meta-model is depicted in Figure 10. The meta-model is used to validate OperA models (as mentioned in the Validation Framework earlier). The OperA models created by OperettA support the specification of Multi Agent Systems (MAS) and can be used in different ways, both during the design phase of MAS as well as at runtime. During the design phase, the OperA model guides the definition of the MAS architecture and provides a basis for the specification of agents and their interactions. In fact, role specifications can be used as a skeleton for agents tailor-made to enact that role. This results in MAS that comply with the organization model described by the OperA specification. However, OperA assumes the separation between organization and agent specification which means that OperA models should be 'interpretable' by agents that have been developed elsewhere. In this sense, OperA models can be used by existing agents at execution time to evaluate their possibilities of enacting that organization. That is, agents could request information about the organization, evaluate how their participation in that organization would benefit their goals and decide to take up a role in the organization, hence accepting the responsibility to fulfill that role's objectives.

The design phase approach uses Model Driven Engineering (MDE) principles based on meta-model transformation. MDE refers to the systematic use of models as primary artifacts throughout the Software Engineering lifecycle. The defining characteristics of MDE is the use of models to represent the important aspect of the system, be it requirements, high-level designs, user data structures, views, interoperability interfaces, test cases, or implementation-level artifacts such as code. The Model Driven Development promotes the automatic transformation

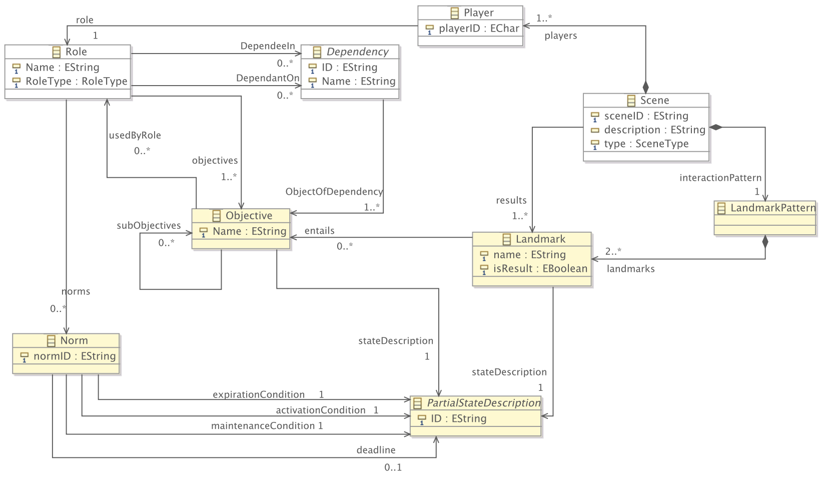


Fig. 10. OperA meta-model (fragment)

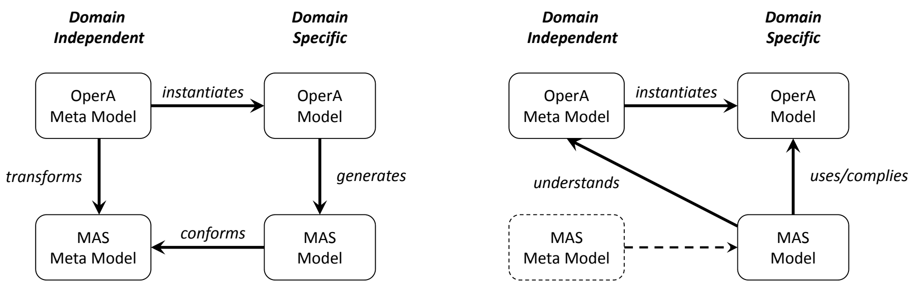


Fig. 11. MDE to generate agents (left) and for use by organisation-aware agents (right)

of abstracted models into specific implementation technologies, by a series of predefined model transformations.

Following this approach, depicted in Figure 11 (left), a transformation is defined between the OperA meta-model and the meta-model of a specific MAS architecture. Using this transformation and a domain specific OperA model, a MAS can be generated that complies to the particular MAS meta-model and implements the organization model defined in the OperA model. This approach was followed in the ALIVE project for the generation of AgentScape systems from OperA models [10]. In the same way, agent-based simulations can be developed.

The execution phase approach requires agents that are able to understand the OperA meta-model such that they can enact roles in an organization, defined as a domain specific OperA model (cf. Figure 11 (right)). Agents who want to enter and play roles in an organization are expected to understand and reason about the organizational specification, if they are to operate effectively and flexibly in the organization. This implies that agents should have reflective capabilities with respect to their own goals, beliefs, perceptions and action potential. Agents that are capable of such organizational reasoning and decision making are called organization-aware agents [20]. In [19] we investigate how GOAL [12] agents can determine whether it has the necessary capabilities to play roles in an OperA organization.

5 Design Guidelines

In the previous we introduced organizational modeling and the OperettA Environment to support this. In this section we present a small overview on how one goes about designing an organization. After identifying that an organization presents the solution to the problem:

1. Identify (functional) requirements: First one determines the global functionalities and objectives of the society.
2. Identify stakeholders: The analysis of the objectives of the stakeholders identifies the operational roles in the society. These first two steps set the basis of the social structure of the OperA model.

3. Set social norms, define normative expectations: The analysis of the requirements and characteristics of the domain results in the specification of the normative characteristics of the society. This results in the norms in the normative structure.
4. Refine behavior: Using *means-end* and *contribution* analysis, a match can be made between what roles should provide and what roles can provide. This aspect contributes to refinement of role objectives and rights.
5. Create interaction scripts: Using the results from steps 3 and 4, one can now specify the patterns of interaction for the organization, resulting in the interaction structure.

More details about the methodological steps taken to create organizational models can be found in [5].

6 Conclusions

In this paper, we present an organization-oriented modeling approach for system development. The OperA modeling framework can be used for different types of domains from closed to open environments and takes into consideration the differences between global and individual concerns. The OperettA tool supports software and services engineering based on the OperA modeling framework. It has been used in the European project ALIVE [10] that combines cutting edge coordination technology and organization models to provide flexible, high-level means to model the structure of inter-actions between services in an environment.

References

1. Aldewereld, H., Álvarez-Napagao, S., Dignum, F., Vázquez-Salceda, J.: Engineering social reality with inheritance relations. In: Aldewereld, H., Dignum, V., Picard, G. (eds.) ESAW 2009. LNCS, vol. 5881, pp. 116–131. Springer, Heidelberg (2009)
2. Dignum, V.: A Model for Organizational Interaction: based on Agents, founded in Logic. SIKS Dissertation Series 2004-1. Utrecht University, PhD Thesis (2004)
3. Dignum, V.: The role of organization in agent systems. In: Dignum, V. (ed.) Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models, pp. 1–16. Information Science Reference (2009)
4. Dignum, V., Dignum, F.: Designing agent systems: State of the practice. International Journal on Agent-Oriented Software Engineering 4(3) (2010)
5. Dignum, V., Dignum, F., Meyer, J.J.: An agent-mediated approach to the support of knowledge sharing in organizations. Knowledge Engineering Review 19(2), 147–174 (2004)
6. Dignum, V., Vazquez-Salceda, J., Dignum, F.: OMNI: Introducing social structure, norms and ontologies into agent organizations. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 181–198. Springer, Heidelberg (2005)
7. Dignum, V., Dignum, F.: Modelling agent societies: Co-ordination frameworks and institutions. In: Brazdil, P.B., Jorge, A.M. (eds.) EPIA 2001. LNCS (LNAI), vol. 2258, pp. 191–204. Springer, Heidelberg (2001)

8. Esteva, M., Padget, J., Sierra, C.: Formalizing a language for institutions and norms. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 348–366. Springer, Heidelberg (2002)
9. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In: ICMAS 1998, pp. 128–135. IEEE, Los Alamitos (1998)
10. European Commission FP7-215890. ALIVE (2009), http://www.ist-alive.eu/
11. Grossi, D., Dignum, F., Dastani, M., Royakkers, L.: Foundations of organizational structures in multiagent systems. In: AAMAS 2005: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 690–697. ACM, New York (2005)
12. Hindriks, K.V.: Programming rational agents in GOAL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Tools and Applications. Springer, Berlin (2009)
13. Kumar, S., Huber, M., Cohen, P., McGee, D.: Towards a formalism for conversation protocols using joint intention theory. Comp. Intelligence 18(2) (2002)
14. Oren, N., Panagiotidi, S., Vázquez-Salceda, J., Modgil, S., Luck, M., Miles, S.: Towards a formalisation of electronic contracting environments. In: Hübner, J.F., Matson, E., Boissier, O., Dignum, V. (eds.) COIN@AAMAS 2008. LNCS, vol. 5428, pp. 156–171. Springer, Heidelberg (2009)
15. Van Dyke Parunak, H., Odell, J.J.: Representing social structures in UML. In: Wooldridge, M., Weiss, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, p. 1. Springer, Heidelberg (2002)
16. Penserini, L., Grossi, D., Dignum, F., Dignum, V., Aldewereld, H.: Evaluating organizational configurations. In: IEEE/WIC/ACM International Conference on Intelligent Agent Technology, IAT 2009 (2009)
17. Smith, I., Cohen, P., Bradshaw, J., Greaves, M., Holmback, H.: Designing conversation policies using joint intention theory. In: ICMAS-1998. IEEE, New York (1998)
18. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Eclipse Series. Addison-Wesley Professional, London (2008)
19. van Riemsdijk, B., Dignum, V., Jonker, C., Aldewereld, H.: Programming role enactment through reflection (2010) (under Submission)
20. van Riemsdijk, M.B., Hindriks, K. V., Jonker, C. M.: Programming organization-aware agents. In: Aldewereld, H., Dignum, V., Picard, G. (eds.) ESAW 2009. LNCS, vol. 5881, pp. 98–112. Springer, Heidelberg (2009)
21. W3C. Owl-s (2004), http://www.w3c.org/Submission/OWL-S
22. Weigand, H., Dignum, V.: I am autonomous, you are autonomous. In: Nickles, M., Rovatsos, M., Weiss, G. (eds.) AUTONOMY 2003. LNCS (LNAI), vol. 2969, pp. 227–236. Springer, Heidelberg (2004)
23. Zambonelli, F.: Abstractions and infrastructures for the design and development of mobile agent organizations. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNAI, vol. 2222, pp. 245–262. Springer, Heidelberg (2002)
24. Zambonelli, F., Jennings, N., Wooldridge, M.: Organizational abstractions for the analysis and design of multi-agent systems. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 235–251. Springer, Heidelberg (2001)

Towards Efficient Multi-agent Abduction Protocols

Gauvain Bourgne[1], Katsumi Inoue[1], and Nicolas Maudet[2]

[1] National Institute of Informatics
Tokyo, Japan
{bourgne,ki}@nii.ac.jp
[2] LAMSADE,
Paris Dauphine University, France
nicolas.maudet@lamsade.dauphine.fr

Abstract. What happens when distributed sources of information (agents) hold and acquire information locally, and have to communicate with neighbouring agents in order to refine their hypothesis regarding the actual global state of this environment? This question occurs when it is not possible (*e. g.* for practical or privacy concerns) to collect observations and knowledge, and centrally compute the resulting theory. In this paper, we assume that agents are equipped with full clausal theories and individually face abductive tasks, in a globally consistent environment. We adopt a learner/critic approach. We present the Multi-agent Abductive Reasoning System (MARS), a protocol guaranteeing convergence to a situation "sufficiently" satisfying as far as consistency of the system is concerned. Abduction in a full clausal theory has however already a high computational cost in centralized settings, which can become much worse with arbitrary distributions. We thus discuss ways to use knowledge about each agent's theory language to improve efficiency. We then present some first experimental results to assess the impact of those refinements.

1 Introduction

In multi-agent systems, the inherent distribution of autonomous entities, perceiving and acting locally, is the source of many challenging questions. To overcome the limitation of their own knowledge, usually local and incomplete, agents are driven to form some hypotheses and share information with other agents. Especially, abductive reasoning is a form of hypothetical reasoning deriving the possible causes of an observation. It can be used to complete an agent's understanding of its environment by explaining its observations, or, more pro-actively, for planning, as one can try to find the possible actions that might cause the completion of a goal. However reasoning in a sound manner with distributed knowledge rises interesting problems, as one cannot ensure locally the consistency of an information. Moreover, the system often comes with severe communication restrictions, due to physical (*e. g.* the limited scope of a communication device) or reasoning (*e. g.* the mere impossibility to consider all the potential communications) limitations of agents populating it. For such situations, we presented in [5] a sound mechanism that is guaranteed to find an abductive hypotheses with respect to distributed full clausal theories whenever one exists. This Multi-agent Abductive Reasoning System, MARS, is based on a consequence finding tools named SOLAR [14], that serves as a main reasoning engine. To be able to use such mechanism in classical agent applications, it is important to define it in term of protocols and strategies. This paper first propose such a description, providing a detailed account of the communications between the agents which was lacking from the previous version. We are then concerned here with the efficiency of this mechanism, and thus want to evaluate and improve its average computational and communicational cost.

Distributed abduction has been considered in recent years in the ALIAS system [6]. They distribute the abductive programming algorithm of [10], using abductive logic program to represent each agent's theory. More recently, DARE [12] addressed a similar problem, but consider possible dynamicity of the system by allowing agents to enter or exit some proof cluster. In none of these works however is the issue of communication constraints explicitely raised. Studies of ALP with runtime addition of integrity constraint [2] is also of interest for distributed abduction as it can be applied to negotiations in which an agent shares its integrity constraints as needed. Another related work is the peer-to-peer consequence finding algorithm DeCA [1]. Based on a different method (splitting clauses), it is to our knowledge the only other work in this domain taking into account restrictions of communication between peers. It is however restricted to propositional theories. The work on partition-based logical reasoning presented [3] is of particular interest for our present study as it investigates efficient theorem proving in partitioned theories. It relies on communication languages describing the common symbol in the individual languages of pairs of agents. However, this approach and the previous one explore all the consequences of the distributed theories, whereas we are only concerned with some *new* consequences of the theories with respect to some knowledge (namely the negated observations when computing a hypothesis through inverse entailment, or the hypothesis itself when ensuring its consistency). As a result, while inspirational to improve the efficiency, they cannot be directly applied to our approach.

The rest of this paper is as follows. Section 2 gives the necessary background on abduction and consequence finding. Then, Section 3 describe formally a multi-agent abduction problem, and present the MARS protocol, giving details about the communications exchanged over its execution. Efficiency is then discussed, and we describe two improvements on the previous protocol. These variants are then experimentally tested in Section 4, and we conclude in Section 5.

2 Abductive Reasoning

2.1 Preliminaries

First, we review some notions and terminology to represent our problem in a logical setting. An *atom* is given as $P(t_1, \ldots, t_n)$ where P is a predicate symbol of arity n and $t_1, \ldots, t_n$ are terms (variables or constants). A *literal* is an atom or the negation of an atom. A *clause* is a disjunction of literals, and is often denoted by a set of literals. A clause $\{A_1, \ldots, A_m, \neg B_1, \ldots, \neg B_n\}$, where A_i and $\neg B_j$ are respectively positive and negative atoms, can also be written as $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$. Any variable in a clause is assumed to be universally quantified at the front. A *Horn clause* is a clause that contains at most one positive literals (the rest being negative literals), otherwise it is *non-Horn*. A *clausal theory* is a finite conjunction of clauses which is usually written as a set of clauses.

Let S and T be clausal theories. S *entails* T, denoted as $S \models T$, if and only if for every interpretation I such that S is true under I, T is also true under I. $\models$ is called the *entailment relation*. For a clausal theory $\mathcal{T}$, a *consequence* of $\mathcal{T}$ is a clause entailed by $\mathcal{T}$. We denote by $Th(\mathcal{T})$ the set of all consequences of $\mathcal{T}$. Let C and D be two clauses. C *subsumes* D, denoted $C \succeq D$, if there is a substitution θ such that $C\theta \subseteq D$[1]. C *properly* subsumes D if $C \succeq D$ but $D \not\succeq C$. For a clausal theory $\mathcal{T}$, $\mu\mathcal{T}$ denotes the set of subsumption-minimal clauses of $\mathcal{T}$, that is the clauses in $\mathcal{T}$ which are not properly subsumed by any clause in $\mathcal{T}$.

We can now introduce the notion of *characteristic clauses*, which represents "interesting" consequences of a given problem [7]. Each characteristic clause is constructed over a sub-vocabulary of the representation language called a *production field*, and represented as $\langle \mathcal{L} \rangle$, where $\mathcal{L}$ is a set of literals closed under instantiation[2]. A clause C *belongs* to $\mathcal{P} = \langle \mathcal{L} \rangle$ if every literal in C belongs to $\mathcal{L}$. For a clausal theory $\mathcal{T}$, the set of consequences of $\mathcal{T}$ belonging to $\mathcal{P}$ is denoted $Th_{\mathcal{P}}(\mathcal{T})$. Then, the characteristic clauses of $\mathcal{T}$ wrt to $\mathcal{P}$ are defined as $Carc(\mathcal{T}, \mathcal{P}) = \mu Th_{\mathcal{P}}(\mathcal{T})$, where, as said before, μ represents subsumption minimality.

When a set of new clauses S is added to a clausal theory, some consequences are newly derived with this additional information. The set of such clauses that belong to the production field are called *new characteristic clauses* of S wrt $\mathcal{T}$ and $\mathcal{P}$; they are defined as $Newcarc(\mathcal{T}, S, \mathcal{P}) = Carc(\mathcal{T} \cup S, \mathcal{P}) \setminus Carc(\mathcal{T}, \mathcal{P})$.

2.2 Abductive Hypothesis

The logical framework of hypothesis generation in abduction for the centralized case can be expressed as follows. Let $\mathcal{T}$ be a clausal theory, which represents the *background theory*, and O be a set of literals, which represents *observations*. Also let $\mathcal{A}$ be a set of literals representing the set of *abducibles*, which are candidate assumptions to be added to $\mathcal{T}$ for explaining O. Given $\mathcal{T}$, O and $\mathcal{A}$, the abduction problem is to find a *hypothesis* H such that:

(i) $\mathcal{T} \cup H \models O$ (accountability),
(ii) $\mathcal{T} \cup H \not\models \bot$ (consistency), and
(iii) H is a set of instances of literals from $\mathcal{A}$ (bias).

In this case, H is also called an *explanation* of O (with respects to $\mathcal{T}$ and $\mathcal{A}$). A hypothesis is *minimal* if no proposer subset of H satisfies the above three conditions (which is equivalent to subsumption minimality for ground clauses). A hypothesis is *ground* if it is a set of ground literals (literals containing no variable). This restriction is often employed in applications whose observations are also given as ground literals. In the following, we shall indeed assume that observations are grounded, and that we are only searching for minimal ground hypotheses.

2.3 Computation through Hypothesis Finding

Given the observations O, each hypothesis H of O can be computed by the principle of *inverse entailment* [7,13], which converts the accountability condition (i) to

[1] Meaning that for all literals of $C\theta$, L_i are also literals of D.

[2] Usually, a production field also includes a condition $Cond$ and is thus written $\langle \mathcal{L}, Cond \rangle$, but we skip it here to simplify the presentation.

$\mathcal{T} \cup \{\neg O\} \models \neg H$, where $\neg O = \bigvee_{L \in O} \neg L$ and $\neg H = \bigvee_{L \in H} \neg L$. Note that both $\neg O$ and $\neg H$ are clauses since O and H are sets of literals. Similarly, consistency condition (ii) is equivalent to $\mathcal{T} \not\models \neg H$. Hence, for any hypothesis H, its negated form $\neg H$ is deductively obtained as a "new" theorem of $\mathcal{T} \cup \{\neg O\}$ that is not an "old" theorem of $\mathcal{T}$ alone. Moreover, to respect the bias condition (iii), every literal of $\neg H$ has to be an instance of a literal in $\bar{\mathcal{A}} = \{\neg L | L \in \mathcal{A}\}$. Then the negation of minimal hypotheses are the new characteristic clauses of O with respect to $\mathcal{T}$ and $\bar{\mathcal{A}}$, that is, $Newcarc(\mathcal{T}, \{\neg O\}, \bar{\mathcal{A}})$.

SOLAR [14] is a sophisticated deductive reasoning system based on SOL-resolution [7], which is sound and complete for finding *minimal* consequences belonging to a given language bias (a *production field*). Consequence-finding by SOLAR is performed by *skipping* literals belonging to a production field $\mathcal{P}$ instead of resolving them. Those skipped literals are then collected at the end of a proof, which constitute a clause as a logical consequence of the axiom set. Using SOLAR, we can implement an abductive system that is *complete* for finding minimal explanations due to the completeness of consequence-finding.

Although many abductive procedures have been implemented by augmenting a resolution based top-down proof procedure [9], they are mainly designed for Horn clauses of normal logic programs. SOLAR is designed for *full clausal theories* containing non-Horn clauses, and is based on a *connection tableau* format [11]. In this format, many redundant deductions are avoided using various state-of-the-art pruning techniques [14], thereby hypothesis-finding is efficiently realized.

Once possible hypotheses have been computed, a ranking process can be applied to select a *preferred hypothesis* (*e.g.* hypothesis ranking such as in [8]). We will not dwell on this part here, and instead assumed that a *preference relation* $\geq_p$ over the hypothesis is given as a total order between sets of grounded literals.

3 Distributed Abduction

3.1 Problem Setting

We propose here a new formalization of our problem as a *multi-agent abductive system*, before defining acceptable solution for such problems, and discussing the relation between local and group consistency and accountability.

Definition 1 (Multi-agent abductive system). *A* multi-agent abductive system *is defined as a tuple* $\langle \mathcal{S}, \{\Gamma_t\}, \mathcal{A}, \geq_p \rangle$*, where:*

- $\mathcal{S} = \{a_0, \ldots, a_{n-1}\}$ *is a set of agents. Each agent* a_i *has its own* individual theory $\mathcal{T}_i$ *and its own* observations O_i. *It will also form its own preferred hypothesis* H_i*, though it can also adopt it from other agents. In fact, in the end of the process, all agents will share the same hypothesis.*
- $\Gamma_t = \langle \mathcal{S}, E_t \rangle$ *is the* communicational constraint graph *at time* t*, an undirected unlabeled graph whose nodes are the agents in* $\mathcal{S}$ *and whose edges* E_t *represent the communicational links between the agent. An agent* a_i *can only communicate with another agent* a_j *at time* t *if* $(a_i, a_j) \in E_t$.

- $\mathcal{A}$ *is the common set of* abducibles *that represents the language bias of the abductive process.*
- $\geq_p$ *is the* common preference relation*, a total order over hypotheses.*

Theories and observations are considered to be certain knowledge. As such, they are assumed to be consistent, meaning that $\bigcup_{i<n} \mathcal{T}_i \cup \bigcup_{i<n} O_i \not\models \bot$. To ensure termination, it will also be assumed that $Carc(\bigcup_{i<n} \mathcal{T}_i, \langle \mathcal{L} \rangle)$ is finite, and that both hypotheses and observations are ground (*i.e.* contain no variable). Moreover, the system will be assumed to be *temporally connected*, meaning that at any time t, the graph $\Gamma_{t^+} = \langle \mathcal{S}, \bigcup_{t' \geq t} E_{t'} \rangle$ is a connected graph.

Problem Statement. Our aim is then to ensure the formation of an abductive explanation of $\bigcup_{i<n} O_i$ with respects to $\bigcup_{i<n} \mathcal{T}_i$ and $\mathcal{A}$. Given a group of agents $G = \{a_i, i \in J\} \subset \mathcal{S}$, we shall say that a hypothesis H is *group-consistent* with G iff it is consistent with the union of all the individual theory of the agents of the group, that is, iff $\bigcup_{i \in J} \mathcal{T}_i \cup H \not\models \bot$. Likewise, we shall say that H ensures *group-accountability* for G iff it can explains all observations of the agents of the group when it is associated with the union of their theories, that is iff $\bigcup_{i \in J} \mathcal{T}_i \cup H \models \bigcup_{i \in J} O_i$. If $G = \mathcal{S}$, we shall say that the hypothesis is *mas-consistent* or that it ensures *mas-accountability*. Finally we shall say that a set of literals is *acceptable* for a group G iff it is a set of grounded literals of $\mathcal{A}$ that is group-consistent with G and ensures group-accountability for G. The objective of a multi-agent abductive system is thus to find a hypothesis that is *acceptable* for the whole system.

While consistency or accountability of a hypothesis with respect to both $(\mathcal{T}_i, O_i)$ and $(\mathcal{T}_j, O_j)$ is not equivalent to consistency or accountability wrt $(\mathcal{T}_i \cup \mathcal{T}_j, O_i \cup O_j)$, we still can ensure some relation between them in classical logic. Specifically, group-inconsistency of H with G implies group-inconsistency of H with any superset of G, which ensures that hypothesis inconsistent with a sub-group of agents (possibly a single agent) can be ruled out as a potential solution. Moreover, group-accountability of H for both G and G' implies group-accountability of H for $G \cup G'$ (but not reciprocally), which ensures that accountability can be checked locally.

In order for a learner agent to propose a hypothesis to a critic, it is necessary that his agent can produce such a hypothesis. However, given only a few clauses of the whole clausal theory, it might not be able to find an explanation for the observations using only abducibles. Therefore, we shall allow an agent to build *partial hypothesis*, which contains some non-abducible literals. Those literals might be the unexplained observations, or preferrably some other literals of the language that would explain it. While interacting with other agents, they will share knowledge to expand these hypotheses in order to progressively build a fully abducible one. Note that of course, a hypothesis respecting the bias condition will always be favored over one who does not.

We shall now present MARS, a mechanism for solving multi-agent abductive problems based on SOLAR.

3.2 Bilateral Interaction

To deal with distributed hypothesis formation in multi-agent systems, we take a learner-critic approach, in which learner agents aim at producing a globally adequate hypothesis through internal computations and local interactions with other agents acting as critics. In our abductive setting, however, critic agent cannot ensure the consistency of the hypothesis by itself, and needs to interact with the learner in order to find incoherence (computing the context of a hypothesis) and produce complete hypotheses (exchanging useful information by justifying partial hypotheses). As the critic agent might have better grounds to built a good hypothesis, we will also allow it to reverse the roles by acting in turn as learner.

The underlying mechanism was presented and proved correct in [5]. Here, we shall introduce the actual protocol based on that procedure, recapitulating its main steps while giving an exact account of the communications involved. Such formalization in term of interaction protocol and strategies should indeed ease the incorporation of this mechanism in other agent system. Provided that agents are equipped with the proper strategies to deal with this protocol, it can then be consistently integrated in classical agent architecture. Our protocol is designed in such a way that whenever an agent sends a message to another, it gets an answer before sending another message. We compactly describe the specifications of the interaction protocol using a statechart [15]. Fig. 1 illustrate the biateral protocol driving local interactions between two agents. Nodes indicate states of the agents (steps of the mechanism), with superscript L or C indicating whether it concerns the *Learner* agent or the *Critic* agent. Note that states 13 and 14 indicate a switching of the roles, as the critic becomes the learner. Labeled arcs indicate that a given message can be sent by an agent in a given state, making the other agent go to the target state upon reception. Dashed arcs indicate an internal change of state without communication. This mechanism is divided in four main steps that we shall now detail.

Hypothesis Selection. An interaction is initiated by a learner agent a_0, in state 1^L, proposing its hypothesis and its validity context to a critic agent a_1 ($propose(H_0)$). If learner's information has changed since it last computed its possible hypotheses, it will recompute them through inverse entailment, using $\bar{\mathcal{A}}$ as a production field. In case it cannot find a hypothesis this way, it will compute a *partial hypothesis* by using an *extended set of abducibles* (possibly the whole language). If the proposed hypothesis h_0 is a new one, the first context Ctx_0 will be computed as the new consequences of $h_0 \cup \mathcal{T}_0$ wrt h_0, that is $Newcarc(\mathcal{T}_0, h_0, \mathcal{P}_\mathcal{L})$ where $\mathcal{P}_\mathcal{L} = \langle \mathcal{L} \rangle$. Otherwise, the previously computed context will be used as initial context Ctx_0. Then, when receiving such proposal, a_1 will start its critic, which consists of three steps: consistency check, accountability check and admissibility check.

As the interaction continues, new hypotheses might have to be proposed. If the current learner cannot propose a hypothesis (which can only happens if it has blocked all its possible hypotheses during previous admissibility checks), it will send $noHyp$ to the other agent to switch the roles (state 14). If this one has also exhausted all its hypotheses, then it will unblock all its hypotheses and propose again the best one ($repropose$). Note that the new critic agent will also unblock all its hypotheses when receiving such a message.

Consistency check. When receiving a proposed hypothesis and context, the first step of the critique is to check the group-consistency of the hypothesis with both agents involved. A context is progressively built for a given hypothesis H to compute the new consequences of $H \cup \mathcal{T}_0 \cup \mathcal{T}_1$ wrt to $\mathcal{T}_0 \cup \mathcal{T}_1$. If the hypothesis is incoherent, then it will

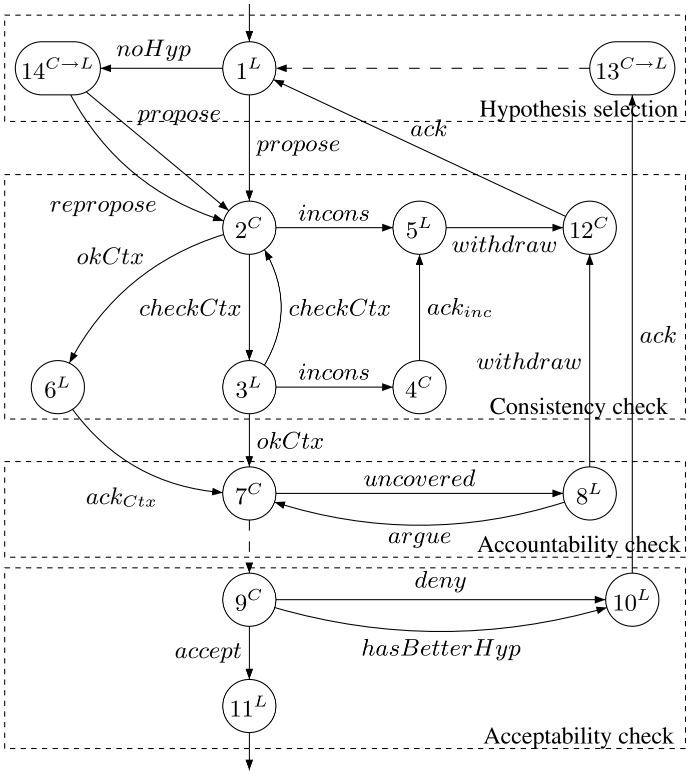


Fig. 1. Multi-Agent Learner-Critic Abductive Protocol

have $\perp$ as a consequence. The occurence of a contradiction between the context and the agents' theories will thus enable detection of incoherent hypotheses. This relies on the fact that the global theory itself is assumed to be consistent, so any inconsistency can only arise from the hypotheses. Indeed, if $\mathcal{T}$ is consistent and $\mathcal{T} \cup H$ is inconsistent, then $Newcarc(\mathcal{T}, U, \mathcal{P}_{\mathcal{L}}) = \{\perp\}$. The process is as follow:

1. First, remember that during the hypothesis selection step, learner agent a_0 retrieve context Ctx_0 of its hypothesis. If no context have been memorized from previous iteraction, then a new one is computed as $Newcarc(\mathcal{T}_0, H_0, \mathcal{P}_{\mathcal{L}})$ (note that we should thus have $H_0 \in Ctx_0$).
2. When receiving H_0 and Ctx_0 (state 2^C), a_1 first check if it already has some context Ctx' for this given hypothesis. If it is the case, then it replaces Ctx_0 by $Ctx'_0 = Ctx_0 \cup Ctx'$. Context Ctx_1 is then computed as $Newcarc(\mathcal{T}_1, Ctx'_0, \mathcal{P}_{\mathcal{L}})$, and sent back to a_0 with message $CheckCtx(Ctx_1)$ (unless a contradiction is found).
3. The process continues. At each step Ctx_i is computed by agent a_α as the new consequences $Newcarc(\mathcal{T}_\alpha, Ctx_{i-1}, \mathcal{P}_{\mathcal{L}})$, where $\alpha = 0$ if i is odd (state 3^L), and $\alpha = 1$ otherwise (state 2^C), and sent with a $checkCtx$ message.

4. This computation stops when either a contradiction is found or Ctx_i is included in either Ctx_{i-1} or Ctx_{i-2}, in which case all new consequences have been computed.
   - If an inconsistency is discovered, the part p_0 of the hypothesis responsible for it is sent to the other agents with message $incons(p_0)$. leading eventually to state 5^L. Both agents rule out p_0 (and any hypothesis containing it) by adding its negation to their theory. The learner agent then move on to its next hypothesis and propose it, trigerring a new critic phase (states 12^C and 1^L).
   - Otherwise, the end of the computation is acknowledged by sending $okCtx$. Both agents memorize the final context $Ctx_f = Ctx_i \cap Ctx_{i-1}$ (where i is the final step) of this hypothesis. Any element in respectively $Ctx_i \setminus Ctx_{i-1}$ and $Ctx_{i-2} \setminus Ctx_{i-1}$ are added to $\mathcal{T}_{1-\alpha}$ and $\mathcal{T}_\alpha$ where again $\alpha = 0$ is i is odd and 1 otherwise. Indeed an element will only be removed from the context if it is a direct consequence of one of the agent's theory. The critic phase move to the next step (state 7^C).

This process guarantees that all new consequences of H_0 with respect to the theories of both the learner and critic agents are computed. This context can then be memorized for future interactions. Indeed our aim is ultimately to check the new consequences of the hypothesis with the union of all theories. Using the previously computed contexts as initialization of this process (by learner and critic) ensures that we can achieve that through series of local interactions in spite of possible communicational constraint (given that the communication graph is temporally connected).

Accountability check. In this step, the critic agent checks if all its observations are explained by $H_0 \cup \mathcal{T}_1$. If an unexplained observation o is found, the message $uncovered(o)$ is sent to the learner agent, now in state 8^L. We then have two possibilities.

- If o is not explained by $H_0 \cup \mathcal{T}_0$, it is a true counter-example. The learner agent then computes a new hypothesis that will also cover o, and proposes it, triggering a new critic phase (states 12^C and 1^L).
- If o is already explained by $H_0 \cup \mathcal{T}_0$, then the learner agent will notify the critic of this fact with $argue(p_0)$, where p_0 is the part of the hypothesis that is used in explaining o with $\mathcal{T}_0$. The critic agent will add the clause $\{o \vee \neg p_0\}$ in its theory[3]. This new information will ensure that the critic agent can find the hypothesis on its own in further steps, or build up upon it. It will then proceed to the next unexplained observation. It is especially useful when no acceptable hypothesis can be found as sharing the rules for forming partial hypotheses might enable the formation of an acceptable one.

If there is no unexplained observation, the critic proceeds to the next step (state 9^C).

Acceptability Check. Any hypothesis that reaches this step is consistent and accounts for the observations, but it might include some non-abducible literals, or unnecessary parts. This step ensures that alternative hypotheses are explored if needed.

1. If the critic has a hypothesis H_c that is prefered to H_0 (according to $\geq_p$), it will reverse roles ($hasBetterHyp$) and submit it. This will finally either result in the

[3] Note that since p is a conjonction of literals, $\neg p_0$ is indeed a clause.

acceptation of a better hypothesis (e.g. if the learner agent lacked the proper rules to build it), or cause the former critic agent to learn why its hypothesis cannot be used (e.g. if the critic agent did not have some observation that should be explained, or if he did not have information invalidating its current hypothesis).

2. Otherwise, if the hypothesis contains non-abducibles (partial hypothesis), the critic agent will temporarily block it, and ask the other agent to do the same ($deny$). I will then also switch roles (state $13^{C \to L}$). This ensures that all partial hypotheses that could provoke information exchange leading to building an abducible hypothesis are explored if needed.
3. If the hypothesis is acceptable, or if a partial hypothesis has been reproposed (meaning the exploration is complete), then the critic send an $accept$ message. The final outcome of the interaction is thus chosen. Hypotheses that were temporarily blocked are unblocked, and the best hypothesis is chosen as the final hypothesis. It is adopted and memorized by both agents, ending the interaction.

3.3 Group of Agents

Each interaction allows the participants to refine their hypotheses and augment their knowledge concerning their consequences. The protocol described before is enough to allow two agents to form a hypothesis that is group-consistent and ensures group-accountability for the pair of agents. Moreover, since all partial hypotheses are explored if needed (and the knowledge to form them shared), it ensures that an acceptable hypothesis for the pair of agents will be found if it can be done with the union of their theories. When more agents are involved, it is possible to chain such interactions to converge towards a consistent state of the system. To take into account possibly variable communication constraints in the system, we propose a rumor-like approach, ensuring the local behaviour and interactions of the agents make the system converges to a state in which all agents have a mas-consistent hypothesis ensuring mas-accountability.

An agent is motivated by the will to ensure it has an explanation with respect to its neighours. As such, it will attempt to have local interactions with them whenever needed to ensure that, memorizing the result of their last interaction with each of their neighbours. In practice, an agent a_i will engage in a local interaction with a neighbour a_j whenever its hypothesis and context (h_i, Ctx^i) differ from those obtained during its last interaction with a_j. The choice of the interlocutor can affect the speed of the process and the redundancy, as illustrated in a dynamic situation in [4], where agents try to abduce the origin of a fire that is spreading in their premises. In that experiment however, the agents shared most of their background knowledge, and could thus use a much simpler protocol for local interactions. For our evaluation, we shall consider that agents chose interlocutors among their neighbours randomly. When a local interaction is proposed to an agent, this one puts the request in a queue, will accept this interaction as soon as it is available.

In [5], this process was proved to be sound, and to guarantee that a solution is found if there is one (which means completeness can be ensured by running the system several time, adding the negation of the final hypothesis after each step). We do not repeat the proofs here, and just give a sketch of it. Soundness of the process is proved by contradiction by proving that if there is an inconsistency, then there would be a finite proof of

$\perp$ from the hypothesis and the theory that would necessarily have been discovered by the consistency (by induction on the length of a minimal proof). Likewise, we can show that a solution will be found if it exists by proving that if there is an acceptable hypothesis, then there would be a finite proof of it, and, unless another acceptable hypothesis is found first, we can build a sequence of partial hypotheses (steps of that proofs) that would lead to sharing the necessary rule for building it locally. Convergence is ensured thanks to the preference relation, ensuring that an agent will only change its hypothesis or context if it is invalidated or if a better one is proposed. Then we can prove that this can only happens a finite number of time given our assumptions, which ensures that the system will eventually stabilize.

Note that, though we chose a rumor-based approach, it would be possible to generalize our bilateral protocol for n agents. There would then be 1 learner and $n-1$ critic at any time. Hypothesis selection would be very similar (with an order between the agent defining which agent should become learner if the current learner ask to change roles). Then during consistency check, the learner would first build a context with the first critic, then propose it to the next one, starting again from the first critic whenever a new consequence would be proposed, until a given context has been proposed to all critic without generating any new consequence. Accountability check can be done in turn by each of the critic (until the hypothesis is deemed consistent and accountable by all of them, or an unaccounted observation is found). As for acceptability check, reaching this step means that the hypothesis is group-consistent and group-accountable. Critic will first check that none of them has a better hypothesis, and then, either deny the hypothesis (if partial) or they will all accept it. Interactions with the critic could be done sequentially or in parallel, but in the later point, the learner agent will act to resolve any concurrency problem. While such a n agent protocol could be interesting in a fully connected system, we consider that such process can be simulated by sequences of local interactions and will thus focus our study on this latter case, which allows more flexibility with communicational constraints.

3.4 Improving Efficiency

The main computational cost of our mechanism lies in the multiple calls to a consequence-finding tools, which is used in the various steps to conduct the logical reasoning, especially for computing possible hypotheses through inverse entailment, computing the context of these hypotheses, and checking their accountability. To improve efficiency, it is thus crucial to reduce as much as possible the computational cost of each of these calls, as well as to reduce their number.

The tools we are using in our implementation, SOLAR, is based on tableaux methods. We assess the computational cost of a call by counting the number of inferences performed during it. Without entering in the details of the procedure, we will discuss here the factors that influence the cost of the computation of $Newcarc(F, \mathcal{T}, \mathcal{P}_{\mathcal{L}})$. The number of inferences is directly related to the number of clauses used in the procedure. Used claused are clauses which can resolved with one of the top clauses (elements of F), or with a consequence of them. Thus, reducing the number of clauses in $\mathcal{T}$ and more importantly in F can both help to reduce the computations steps. Note that $Carc(\mathcal{T}, \mathcal{P}_{\mathcal{L}})$ is in practice computed as $Newcarc(\mathcal{T}, \emptyset, \mathcal{P}_{\mathcal{L}})$, so computing new consequences rather than all consequences is already a good step to ensure better efficiency. Reducing the number of literals of the top clauses also helps as it limits the number of clauses they can be resolved with. Then, another factor that can affect the computations is the size of the production field. A small production field limits the number of options to be explored and as a results, the number of inferences to be done.

With respects to our mechanism, these considerations means that we should keep each agent's individual theory as small as possible, which is ensured by adding single clauses with just the necessary parts of the hypothesis to memorize inconsistencies (when sending or receiving $incons.(p_0)$ or accountability arguments (when sending $argue(o \vee \neg p_0)$ in state 8^L). Moreover, during consistency check, we should minimize the computations for the context. We shall see in next subsection how to reduce size of top clauses during this step by doing incremental computations. Then, we should also find ways to minimize the number of consequences computed during this consistency step by focusing on consequences that could lead to a contradiction, and even more importantly, to limit the number of partial hypotheses computed by focusing on those partial hypotheses that could trigger information exchanges leading to the formation of an acceptable hypothesis (as it would also reduce the number of applications of SOLAR).

3.5 Incremental Consistency Check

During consistency check, context is progressively computed until it does not evolve anymore, but sending the whole context at each step of the computation and using it as top clause for computing the next step. To avoid redundant communications and computations, we propose to communicate only the new consequences of the context, pruning consequence discovered in previous step. It does not change the computation and sending of Ctx_0 and Ctx_1, but after computing Ctx_1, the learner agent will only send back $ctxStep_1 = Ctx_1 \setminus Ctx_0$ (note that we use the original Ctx_0 here, and not Ctx'_0).

Then, when receiving $checkContext(ctxStep_i)$, agent a_α first computes $NC_{i+1} = NewCarc(ctxStep_i, \mathcal{T}_\alpha \cup Ctx_{i-1})$. It can then use it to compute the current context $Ctx_{i+1} = Ctx_{i-1} \cup NC_{i+1}$, and send the update $ctxStep_{i+1} = NC_{i+1} \setminus ctxStep_i$. If there is a clause c than is in $ctxStep_i$ that is not subsumed by any clause of NC_{i+1}, it means that it is a consequence of $\mathcal{T}_\alpha$. It should then be sent to the other agent (with message $inform(c)$) to ensure that both agents will have the same final context. This replaces the theory adjustment with $Ctx_i \setminus Ctx_{i-1}$ and $Ctx_{i-2} \setminus Ctx_{i-1}$ that were made before. Note that the termination condition becomes much simpler, as the context can be confirmed as soon as $ctxStep_i$ is empty.

3.6 Language Focus

Languages. We first introduce some vocabulary and notations to depicts the differents languages that can be useful to describe the system. Given a clausal theory T, we denote by $L(T)$ the set of non-logical symbols that occur in T, and by $\mathcal{L}(T)$ the language formed upon them. Each agent has its own theory $\mathcal{T}_i$, from which we can define its *individual language* $\mathcal{L}(\mathcal{T}_i)$. We can then compute for each pair of agent a_i, a_j ($i \neq j$), in the manner of [3], the *communications language* $\mathcal{L}_{i,j} = \mathcal{L}(\mathcal{T}_i) \cap \mathcal{L}(\mathcal{T}_j)$. It can be used to direct the focus of bilateral communications. If it is empty, a_i and a_j do not need to communicate together. However, it may be the case that a_i and a_j are never connected while $\mathcal{L}_{i,j}$ is not empty. For the sake of simplicity we will assume that the communicational links are static [4]. We shall then adapt the communications languages by choosing a minimal path for all such pair of unconnected agents (a_i,a_j), and add the $\mathcal{L}_{i,j}$ to the communication language of each pair of agent in this path. In the following, when referring to the communication language $\mathcal{L}_{i,j}$, we will assume that this modification has been done. Then the *restricted individual language* of an agent a_i is defined as $\mathcal{L}_i = \bigcup_{j \in N_i} \mathcal{L}_{i,j}$, where N_i is the set of the indexes of the neighbours of a_i. At last, the *common language* is the language $\mathcal{C} = \bigcup_{i<n} \mathcal{L}_i$, that is the union of all restricted individual languages (which is also .the union of all the communication languages).

Context Narrowing. When computing context, we want to ensure that any new consequence of the hypothesis that can be derived from the union of the agent's theories is indeed found. It means a_0 need to send any consequence of H wrt $\mathcal{T}_0$ that could resolved with a clause of $\mathcal{T}_1$. In pratice, a_0 compute its context using its *restricted individual language* as a production field, and then retain in Ctx_0 only the one that contains at least a literal of the concerned communication language (here $\mathcal{L}_{0,1}$). Upon receiving it, a_1 will then temporarily add $\mathcal{L}(Ctx_0)$ to $\mathcal{L}_{0,1}$, and if it already has a context Ctx', a_1 will use the same pruning before adding it to get Ctx'_0 and compute Ctx_1 (with his restricted individual language as a production field). The pruning (with updated language) is applied to Ctx_1 (or $ctxStep_1$) before sending it, and the process continue. Each time, contexts are pruned to exclude any clauses that do not have literals in the current communication language before being sent to the other agent. Note that since we compute only new consequences, we cannot directly use the communication language as a production field,as shown by the following example:

Example 1. Let's take $\mathcal{T}_0 = \{\neg h \vee a \vee b, \neg h \vee o\}$, $\mathcal{T}_1 = \{\neg a\}$ and $\mathcal{T}_2 = \{\neg b\}$, all agents being connected. We have $\mathcal{L}_{0,1} = \mathcal{L}(\{a\})$, and $\mathcal{L}_{0,2} = \mathcal{L}(\{b\})$, so $\mathcal{L}_0 = \mathcal{L}(\{a, b\})$. We assume a_0 has observation o and wants to check hypothesis h with a_1. If Ctx_0 was computed with $\mathcal{L}_{0,1}$, it would be empty, and no contradiction would be found when a_0 checks later with a_2. However, using $\mathcal{L}_0$ as a production field, we get consequence $a \vee b$ that contains literal $a \in \mathcal{L}_{0,1}$. It is thus sent to a_1 that will give in return b. When proposing this context to a_2 later on, a_0 will thus be able to derive a contradiction.

Choice of Partial Hypotheses. For computing partial hypotheses, and deciding whether to propose a given one to a neighbour or not, the same principles can be used. When no admissible hypothesis can be found, inverse entailment is performed again with an extended set of abducibles. To ensure that at least one solution can be found, the manifestations are added to the abducibles, enabling trivial explanations for some part of the hypothesis. Then, the idea is to use literals that can act as links between the theories. In practice, it means that we should include in the extended abducibles the literals in the restricted individual language of the agent. This should also be augmented with

[4] Otherwise, since the system is assumed to be temporally connected, it is possible to find a connected subgraph that is included in Γ_{t+} for all t and use it as a guaranteed basis.

literals obtained through the arguments of other agents (when receiving $argue(o \vee p_0)$ in state 7^C). This allow us to compute all potentially useful partial hypothesis. For a given exchange, however, it is sufficient to propose those partial hypotheses that contains at least one literal of the communication language of the interacting agents.

4 Experiments

We describe here preliminary experimental results on a set of different problems, testing our two improvements of the MARS protocol (namely, incremental context computation and language focus). Though it might be useful to assert the validity of our conclusions on a broader number of problems, we believe that the small problems used for evaluation highlight the main difficulties that can be encountered in a distributed abduction system. We first give a case study on a given problem for which we vary the topology and number of agents, while keeping regularities in the distribution. Then we describe a few other test problems, and analyze the results for computational and communicational efficiency.

4.1 Case Study

We first describe a case study with a class of propositional problem designed to show a kind of worst case for distribution : the `chain_`n problems. It consists of three chains of implications linking respectively h_1 to o_1 (through $k_{n-2}, \ldots, k_0$), h_1 to o_2 (through $m_{n-2}, \ldots, m_0$) and h_2 to o_1 (through $l_{n-2}, \ldots, l_0$). Moreover, one agent (agent $a_{n/2}$) has a constraint $\neg o_1 \vee \neg o_2$, which makes h_1 inconsistent. The aim is then to explain o_1 with abducibles $\{h_1, h_2\}$. Each agent knows 3 rules, one from each chain, with a different offset for each: $k_{i[n]} \vee \neg k_{(i-1)[n]}$, $l_{(i+1)[n]} \vee \neg l_{i[n]}$ and $m_{(i+2)[n]} \vee \neg m_{(i+1)[n]}$ where $x[n]$ represents x modulo n and k_{n-1}, l_{n-1} and m_{n-1} are interpreted as respectively o_1, o_1, o_2 if positive, or h_2, h_1, h_1 if negative. Moreover, agent a_0 initially has observation o_1.

We shall use this problem with $n = 8$ to study the behaviour of the protocol and the impacts of the presented improvements when the theories of the agents are really mixed together. Figure 2 depicts this problem. Nodes are literals, and each arrow corresponds to a rule, labeled by the agent who has it in its own theory. This problem was tested with either a line topology (from a_0 to a_7) or a circuit topology (as the line, with an additional link between a_0 and a_7). To check the influence of the number of agent, we also used a version of this problem with 4 agents, `chain_8.4`, in which a_0, a_1, a_2, a_3 are merged with respectively a_4, a_5, a_6 and a_7, and a version with 2 agents, `chain_8.2`, in which a_0 are merged with respectively even and odd indexed agents.

With 8 agents in a line and problem `chain_`n, we get the following unfolding. First, a_0 has no way to explain o_1, and this observation is propagated by local interactions to agent a_6. Each of these interaction can only propose o_1 as a partial hypothesis, and o_1 is given to each agent in the line during accountability check as they counter-propose the empty hypothesis. Then a_6 gets observation o_1, and can start proposing partial hypothesis l_6. This partial hypothesis is propagated back to agent a_0, and at each interaction, by justifying its partial hypothesis l_k, the learner a_k allows the critic a_{k-1} to build partial hypothesis l_{k-1}. During this propagation of l_k, if language focus is not used, each agent

$$\begin{tikzcd}
& l_6 \arrow[dl, "a_6"'] & l_5 \arrow[l, "a_5"'] & l_4 \arrow[l, "a_4"'] & l_3 \arrow[l, "a_3"'] & l_2 \arrow[l, "a_2"'] & l_1 \arrow[l, "a_1"'] & l_0 \arrow[l, "a_0"'] & \boxed{h_2} \arrow[l, "a_7"'] \\
\circled{o_1} & & & & & & & & \\
& k_6 \arrow[ul, "a_7"] & k_5 \arrow[l, "a_6"'] & k_4 \arrow[l, "a_5"'] & k_3 \arrow[l, "a_4"'] & k_2 \arrow[l, "a_3"'] & k_1 \arrow[l, "a_2"'] & k_0 \arrow[l, "a_1"'] & \\
& & & & & & & & \boxed{h_1} \arrow[ul, "a_0"'] \arrow[dl, "a_6"] \\
o_2 & m_6 \arrow[l, "a_5"'] & m_5 \arrow[l, "a_4"'] & m_4 \arrow[l, "a_3"'] & m_3 \arrow[l, "a_2"'] & m_2 \arrow[l, "a_1"'] & m_1 \arrow[l, "a_0"'] & m_0 \arrow[l, "a_7"'] & \\
& & & & o_1 \arrow[r, "a_4"] & \neg o_2 & & &
\end{tikzcd}$$

Fig. 2. Chain_8 problem

has to propose every partial hypothesis l_6 to l_k, whereas language focus allows the agent to forego the proposal of useless partial hypotheses. Then, a_0 can propose partial hypothesis l_0. Since a_7 also has l_0 in its theory, this partial hypothesis has to be proposed to it also, and is conveyed to a_7 through a series of interactions with all agents in the line. Concurrently, each time a new partial hypothesis is discovered, unless language focus is used to avoid it, it has to be propagated also toward a_7. In any case, a_7 gets o_1 while l_k partial hypotheses are explored, and start a new series of partial hypothesis starting with k_6. This eventually reach a_0 who can thus at last make hypothesis h_2 and starts checking its consistency with the other agents. Around the same time, the fact that l_0 can explain o_1 reaches a_7 and allows the proposition of h_1, and the computation of its context. Then, while h_2 is propagated from a_0 towards a_7, h_1 is propagated from a_7 towards a_0, with their context growing at each step. When the two hypotheses meet in some interaction, h_1 is favored because of the preference relation, and continues to spread towards a_0. Finally h_1 is proposed to a_0 with a context taking into account input form all other agents, and a contradiction is found. h_2 becomes the new hypothesis, and is spread again towards a_7 with the contradiction of h_1. At last, h_2 reaches a_7 and its context is complete. It is propagated back to a_0 without modification along the way, and the system is then homogeneous. The process stops with the correct hypothesis.

As we can see from this unfolding, we have several kinds of propagation: propagation of new partial hypotheses, as information shared lead to new proposals which are shared with all agents including them in their communication language (if the agents are not connected, such hypothesis are propagated to them along the path in which the relevant literal was added to the communication language), propagation of interesting hypotheses with progressive growth of their associated contexts, and of course, propagation of discovered inconsistencies. Each time an hypothesis (partial or not) is proposed, a context is computed for it, and it is thus helpful to reduce the cost of this computation (though language restriction and incremental context). But restricting the proposition of partial hypotheses to cases in which it might lead to a new one is also quite important, as it can entirely avoid this step and the propagation to the whole system of irrelevant hypotheses. Thus, if there is an acceptable hypothesis, it is better to propagates partial hypotheses, together with the rule allowing their formation, only to the agents who can use this information to build a better hypothesis.

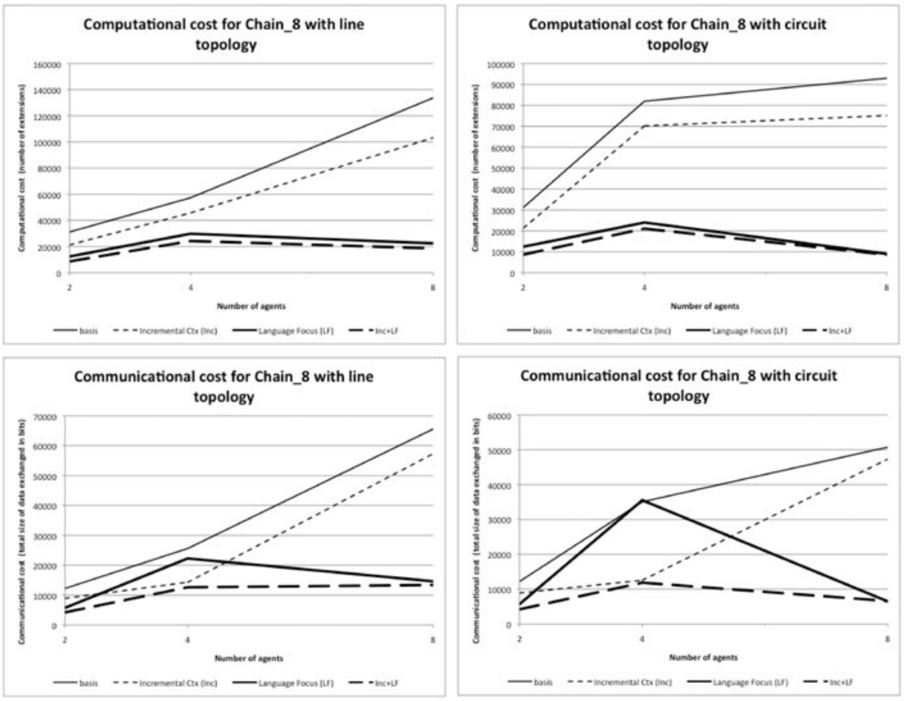


Fig. 3. Efficiency results for Chain_8 with different number of agents and topology

This advantage of language focus is clearly visible on the results for all these problems (see fig 3). In this problem, incremental context computations is also visibly more efficient, though the ratio is not as high. In term of communications, incremental context computations greatly improves performances in almost all cases. One good point is that those two improvements can benefit from being used together, as indeed, in almost the cases presented here, the best performances are given by using both improvements (for both communications and computations).

The distribution and the topology of the communicational constraints also affect the efficiency, but not in a straightforward way. For 8 agents, adding the link between a_0 and a_7 clearly helps, as we saw in the unfolding that several information would otherwise have to be propagated at some cost from one extremity to the other. While a lot of computations and communications are avoided by this, the link between a_0 and a_7 also means that some non-crucial informations might be propagated from a_0 to a_7 while it would not have happened otherwise, especially when language focus is not used. This causes some redundancy as an hypothesis will be checked along more paths. This overheard depends on the structure of the theory, but might counter balance the benefits of avoiding propagation critical information along a long path. This is a priori what happens with 4 agents. In fact a more connected topology favors the quick discovery of an acceptable hypothesis (avoiding the redundancies of exploring each partial hypothesis and propagating an important partial hypothesis through numerous interactions), but in the same time, it allows more redundancies in computations and communications while building contexts as it might for instance propagate for longer an hypothesis that is discovered to be inconsistent in some distant interaction. Using a more controlled way of chaining the local interactions might avoid some of those redundancies, but it would require to choose a single first learner to initiate the sequence instead of relying on a anytime self stabilizing behaviour.

Likewise, the number of agents is not a key factor for efficiency if language focus is used. In this problem, the size of the global theory stays the same, and the number of agents only impacts the way knowledge is distributed. While more agents seems to imply more communications, a proper distribution combined with language focus can keep the communications in relatively low levels and be computationally efficient. In the original problem, each agent is close to another that owns complementary rules, and language focus performs well in directing the computations. The results for 4 agents shows that merging several agents might it fact worsen both the computational and communicational efficiency.

4.2 Other Problems

We also tested our methods with some other different problems, chose to ensure a variety of difficulties.

The first problem, `pb-1`, is taken from [5], where it was used as a running example. It contains 10 clauses, some of them non-Horn, distributed among 2 or 3 agents. With 3 agents, we tested two communicational constraint topologies: a line ($a_0 \leftrightarrow a_1 \leftrightarrow a_2$) and a completely connected system. This problem was designed to illustrated the MARS protocol, and thus make it go throught all the possible states during its enfolding, which ensures that it is both small and not trivial. It is given by:

$\mathcal{T}$=$\{r_1 : \neg g(X) \vee b(X), r_2 : \neg a(c1), r_3 : \neg k(X) \vee b(X), r_4 : \neg h(X) \vee e(X), r_5 : f(X,Y), \neg h(Y) \vee \neg g(X), r_6 : \neg e(X) \vee \neg c(X), r_7 : a(X) \vee c(X) \vee \neg d(Y,X), r_8 : k(X) \vee \neg i(X), r_9 : h(c1), r_{10} : \neg f(X,Y) \vee d(X,Y)\}$

with only one observation $o_1 : [b(c2)]$ given to agent a_0 in all distributions, and abducibles are $i(X)$ and $g(X)$. These clauses are divided among 2 agents as follows : $\mathcal{T}_0^{2ag} = \{r_1, r_2, r_3, r_4, r_5\}$ and $\mathcal{T}_1^{2ag} = \{r_6, r_7, r_8, r_9, r_{10}\}$. The distribution among 3 agents is given by $\mathcal{T}_0^{3ag} = \{r_2, r_7, r_8, r_{10}\}$, $\mathcal{T}_1^{3ag} = \{r_1, r_4, r_5\}$ and $\mathcal{T}_2^{3ag} = \{r_3, r_6, r_9\}$.

The second problem, `pb-fvar`, is a first order toy problem with two observations and 34 clauses, that contains some clauses with unlinked variables. It also contains non-Horn clauses. We pruned out hypotheses that contains variables in the resolution to respect our language bias. It is distributed among 3 agents, and here again, we tested it with line and completely connected graph topologies. We give below the theories of each agents. Note that they all possess 3 rules in common (denoted by $\mathcal{T}_C$).

$\mathcal{T}_C$=$\{\neg hyp(1,X,Y) \vee \neg hyp(3,Y,X), fact(5,1,2,1), fact(4,5,1,5)\}$.

$\mathcal{T}_0 = \mathcal{T}_C \cup \{ \neg o(X) \vee r(X), \neg m(X,Y) \vee \neg n(Y,X) \vee o(X), \neg hyp(3,X,Y) \vee m(X,Y), \neg hyp(1,X,Y) \vee p(X,Y), \neg hyp(2,X,Y) \vee n(Y,X), \neg p(X,Y) \vee \neg q(Y) \vee r(X), \neg fact(Y,2,X,X) \vee q(Y), \neg fact(X,Y,X,Y) \vee m(X,Y) \}$.

$\mathcal{T}_1 = \mathcal{T}_C \cup \{ \neg hyp(3,X,X) \vee g(X), \neg m(X,Y) \vee i(Y) \vee l(X), \neg b(X,Y,X) \vee \neg e(X,Y), \neg g(X) \vee \neg i(Y) \vee e(X,Y), \neg e(X,Y) \vee \neg l(Z) \vee f(X,Y,Z), \neg f(X,6,X) \vee \neg fact(4,Y,X,X), \neg fact(X,3,Y,Z) \vee \neg i(Y) \vee$

$b(X, Z, Y), \neg fact(1, X, Z, X) \vee l(X), fact(1, 2, 5, 2), fact(4, 2, 6, 6), fact(2, 3, 6, 3)$ }.
$\mathcal{T}_2 = \mathcal{T}_C \cup \{ \neg hyp(1, Y, Y) \vee c(Y), \neg hyp(2, X, Y) \vee d(X, Y), \neg d(X, X) \vee \neg l(X), \neg d(X, Y) \vee \neg c(Y) \vee b(X, Y, X), \neg b(X, Y, X) \vee a(Y, X), f(X, Y, X) \vee \neg a(Y, X), \neg fact(X, Y, Z, X) \vee c(Y), \neg fact(2, 3, X, Z) \vee d(X, X), fact(6, 4, 5, 6), fact(2, 3, 3, 5) \}$.
Observation $o_1 : r(6)$ is given to agent a_0, and observation $o_2 : a(3, 6)$ is given to agent a_2. Abducibles are literals $hyp(N, X, Y)$.

At last, we used a more practical problem, schedule$_{var}$, which is an adaption from a scheduling problem presented in [12], with 8 agents. schedule$_{dir}$ is a direct translation of the same problem from its original formalization as an abductive logic program (negation by default is dealt with by using additional abducibles). We give below the theories of schedule$_{var}$:

$\mathcal{T}_{Convener} = \{conveneMeeting(date) \vee \neg day1(D) \vee \neg ansDay(D) \vee \neg tutorName(T) \vee \neg ansTutor(T) \vee \neg lecturerName(L) \vee \neg ansLecturer(L) \vee \neg studentName(S) \vee \neg ansStudent(S), day1(tuesday), day1(wednesday), day1(thursday), day1(friday) \}$.
$\mathcal{T}_{TutorPat} = \{tutorName(pat), \neg ansDay(T) \vee \neg ansTutor(pat) \vee \neg busy2(T), busy2(X) \vee nursery(X), \neg ansTutor(pat) \vee \neg ansStudent(dan), \neg ansTutor(pat) \vee \neg ansLecturer(joe) \}$.
$\mathcal{T}_{StudentBen} = \{\neg ansDay(T) \vee \neg ansStudent(ben) \vee \neg busy3(T), studentName(ben), busy3(tuesday), busy3(wednesday) \}$.
$\mathcal{T}_{StudentDan} = \{\neg ansDay(T) \vee \neg ansStudent(dan) \vee \neg busy4(T), studentName(dan), busy4(tuesday), busy4(thursday), busy4(friday) \}$.
$\mathcal{T}_{Nursery} = \{\neg nursery(monday), \neg nursery(tuesday), nursery(wednesday), \neg nursery(thursday), nursery(friday) \}$.
$\mathcal{T}_{LecturerJoe} = \{lecturerName(joe), \neg ansDay(T) \vee \neg ansLecturer(joe) \vee \neg hasTeaching(T, joe) \}$.
$\mathcal{T}_{Timetabler} = \{hasTeaching(T, N) \vee \neg teachingJuniors7(T, N), hasTeaching(T, N) \vee \neg teachingSeniors7(T, N), teachingSeniors7(thursday, joe), teachingJuniors7(wednesday, rob)\}$.
$\mathcal{T}_{LecturerRob} = \{lecturerName(rob), \neg ansDay(T) \vee \neg ansLecturer(rob) \vee \neg hasTeaching(T, rob), \neg ansDay(T) \vee \neg ansLecturer(rob) \vee \neg tired8(T), tired8(thursday) \}$.

4.3 Results

Tables 1 gives the results for the four variants of our mechanism, while figure 4 gives a more graphical view of the computational results. Computational cost is given by the total number of operations performed by the consequence finding tool over the course of the protocol, whereas communicational cost is expressed as the total number of bits exchanged by the agents during the process. ¿From these results, it is obvious that using individual communication languages does indeed greatly reduce both costs It is especially true for the most complex problems, and the gain ratio is more important when there are a greater number of communicational links. Incremental computation of context is however in the end less convincing, as it only helps when there are several context computations step, which is not such a common occurence, unless theories are really mixed (it is the case for pb-1 and all chain_8 problems, which do benefit from this improvement). In the end, this improvement is useful, but only marginally so in situations where the knowledge of the agents is not heavily mixed (for example in case when agents do not share most of their literals together). While more experiment would be required to say anything more definite, the present results give us some hint about the

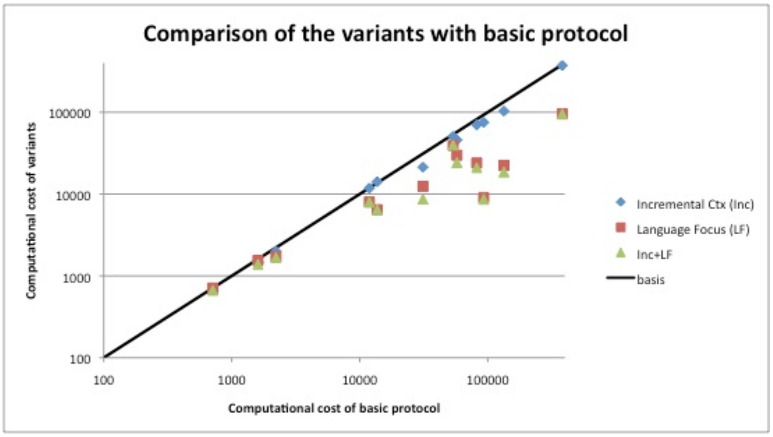


Fig. 4. Comparison computational efficiency of the basic protocol with its variants on 12 problems (logarithmic scale). Each dot represent a problem, and it is below the solid line if the variant it represents performs better than the original protocol on this problem.

influence of topology and "encoding". As discussed in the case study, having a topology with cycle can lead to redundant computations, but can also provide easier exchange of information by avoiding the extra cost of bringing back a crucial fact or rule. Overall, using individual communication language allows us to reduce the cost of redundant computations, so that we can take more benefit from situations where additional links are helpful. Our conjecture is that the cost is lower when the topology is scarce while still ensuring that 2 agents sharing common litterals are not too far apart. Moreover, reducing the number of agents can be either detrimental (in `chain_8`) or benificial (in `pb-1`): the size of the communication languages seem to be a more relevant factor.

At last, the huge difference between $\texttt{schedule}_{var}$ and $\texttt{schedule}_{dir}$ seems to indicate that our protocol is much more efficient for finding whether an abducible hypothesis is consistent than it is for finding an abducible hypothesis by exploring all partial hypotheses. When formalizing a given problem, it is thus more efficient to ensure one agent can easily generate candidate hypotheses, and express rules that constrain

Table 1. Experimental results

		Computations				Communications			
Language focus		no	no	yes	yes	no	no	yes	yes
Incremental ctx comp.		no	yes	no	yes	no	yes	no	yes
Pb-1	2 ag.	711	666	711	666	617	552	617	552
Pb-1 (line)	3 ag.	1 602	1 454	1 548	1 390	1 832	1 700	1 624	1 511
Pb-1 (clique)	3 ag.	2 216	2 003	1 713	1 673	2 540	2 373	2 115	2 064
Pb-fvar (line)	3 ag.	11 890	11 818	8 019	7 959	3 177	3 080	2 659	2 622
Pb-fvar (clique)	3 ag.	13 680	14 126	6 457	6 415	3 568	3 494	2 139	2 106
Chain_8.2	2 ag.	31 171	21 330	12 438	8 675	12 230	8 924	5 746	4 238
Chain_8.4 (line)	4 ag.	57 406	45 803	29 844	24 285	25 606	14 312	22 278	12 620
Chain_8.4 (circ.)	4 ag.	81 998	70 168	23 999	21 075	35 072	12 576	35 531	11 883
Chain_8 (line)	8 ag.	133 696	103 219	22 450	18 600	65 582	57 305	14 625	13 383
Chain_8 (circ.)	8 ag.	92 986	75 146	9 015	8 639	50 749	47 317	6 509	6 627
Schedule_{var}	8 ag.	53 607	50 571	39 391	39 725	28 774	27 171	20 448	20 752
Schedule_{dir}	8 ag.	381 992	372 998	95 909	94 963	219 335	209 398	77 638	76 238

it. Current implementation should however be made more efficient for exploring partial hypotheses. Keeping a more detailed memory of previous interactions should allow to prune a lot of redundant computations for an increased memory cost. Then, current information sharing is not well-suited for dealing with sub-goals (when a observation can be explained by a conjunction of causes, each of them having causes known by different agents). An interesting lead would thus be to replace the empirical argumentation $o \vee \neg p_0$ by a slightly more complex proof using the literals of the communication languages as intermediate conclusions.

5 Conclusion

We presented in this paper a formalization of a multi-agent abduction problem, and proposed a sound mechanism for computing an abductive explanation that is guaranteed to find a solution whenever one exists. We then discussed way to improve the average efficiency of this protocol, called Multi agent Abductive Reasoning System (MARS). Two improvements were proposed. The first one reduce the costs of building a complex context by doing the computation incrementally. It only helps when several steps are needed and was therefore shown to have only a limited impact on efficiency by experimental results. The main improvement consist of using informations about the individual language of each agent to focus the exchanges on what can really advance the search for a hypothesis (or the inconsistence of a candidate hypothesis). Contrarily to [3], we do not need the communication graph to be made cycle-free. While our approach use a similar idea of using *communication language*, we are only interested in the new consequences of some formulas, and thus want to avoid computing all consequences of a theory. As a result, we showed that we needed to allow the exchanges of clauses that belong only partially to the communication language. Nonetheless, it is still an important efficiency improvement compared to the more naive approach of using only the common language for all exchanges. Experimental results showed that it substantially reduces the number of computations as well as the size of the communications. More improvements should however be brought to the search of hypotheses that can only be produced by using the theories of several agents. The learner-critic assumption that hypothesis are produced locally might be unadapted in such situations. It might thus be better to design a collaborative hypothesis formation, though another lead could be to refine the current information exchange to ensure a better treatment of "sub-goals".

References

1. Adjiman, P., Chatalic, P., Goasdoué, F., Rousset, M.-C., Simon, L.: Distributed reasoning in a peer-to-peer setting: Application to the semantic web. J. Artif. Intell. Res (JAIR) 25, 269–314 (2006)
2. Alberti, M., Gavanelli, M., Lamma, E.: Runtime addition of integrity constraints in an abductive proof procedure. In: Hermenegildo, M.V., Schaub, T. (eds.) ICLP (Technical Communications), LIPIcs, vol. 7, pp. 4–13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)

3. Amir, E., McIlraith, S.A.: Partition-based logical reasoning for first-order and propositional theories. AI 162(1-2), 49–88 (2005)
4. Bourgne, G., Hette, G., Maudet, N., Pinson, S.: Hypotheses refinement under topological communication constraints. In: AAMAS, IFAAMAS, p. 239 (2007)
5. Bourgne, G., Inoue, K., Maudet, N.: Abduction of distributed theories through local interactions. In: Proc. of the 19th European Conference on Artificial Intelligence, ECAI 2010 (August 2010)
6. Ciampolini, A., Lamma, E., Mello, P., Toni, F., Torroni, P.: Cooperation and competition in ALIAS: a logic framework for agents that negotiate. Annals of Math. and AI 37(1–2), 65–91 (2003)
7. Inoue, K.: Linear resolution for consequence finding. Artif. Intell. 56(2-3), 301–353 (1992)
8. Inoue, K., Sato, T., Ishihata, M., Kameya, Y., Nabeshima, H.: Evaluating abductive hypotheses using an em algorithm on bdds. In: Proc. of IJCAI 2009, pp. 810–815 (2009)
9. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In: Handbook of Logic in Artificial Intel. and Logic Progr., vol. 5, pp. 234–324. Oxford Univ. Press, Oxford (1998)
10. Kakas, A.C., Mancarella, P.: Database updates through abduction. In: Proc. of VLDB 1990, pp. 650–661. Morgan Kaufmann Pub., San Francisco (1990)
11. Letz, R., Mayr, K., Goller, C.: Controlled integration of the cut rule into connection tableau calculi. JAR 13, 297–338 (1994)
12. Ma, J., Russo, A., Broda, K., Clark, K.: DARE: a system for distributed abductive reasoning. JAAMAS 16-3, 271–297 (2008)
13. Muggleton, S.: Inverse entailment and progol. New Generation Comput. 13(3&4), 245–286 (1995)
14. Nabeshima, H., Iwanuma, K., Inoue, K.: Solar: A consequence finding system for advanced reasoning. Autom. Reas. with Analytic Tableaux and Rel. Meth., 257–263 (2003)
15. Odell, J.J., Van Dyke Parunak, H., Bauer, B.: Representing agent interaction protocols in UML. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 121–140. Springer, Heidelberg (2001)

Validation of Agile Workflows Using Simulation

Kai Jander, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{jander,braubach,pokahr,lamersd}@informatik.uni-hamburg.de

Abstract. Increasing automation of business processes and industrial demand for complex workflow features have led to the development of more flexible and agile workflow concepts. One of those concepts is the use of goal-oriented workflows, which rely on ideas derived from agent technology like describing the workflows based on a goal hierarchy. While this reduces the gap between business view and IT view and allows for easy implementation of contengencies, the concepts have greater conceptual abstraction obscuring the control flow and reducing the ability of workflow engineers to identify specification flaws in the workflow. This paper shows an approach to address this problem by presenting a system for testing and validating workflows within a specified parameter space. The system allows the definition of test cases (scenarios), each of which contains parameter states applied during workflow execution. The workflow engineer can define a set of scenarios for a workflow testing specific situation that are likely to occur during operation or are otherwise interesting corner cases, allowing automated tests and correction of faults before deployment of the workflow in production environments.

1 Introduction

Business process management (BPM) is considered a very promising strategy that helps in aligning companies towards effective and efficient business operation [1]. This is mainly achieved by completely thinking in terms of processes, which means that even the structure of organizations has to follow the processes and cannot be kept in a functional orientation. The vision of BPM assumes that processes can at least partially be automated, monitored according to key performance indicators (KPI) and continually improved or even renewed according to the measurement and defined KPI targets. It becomes clear that this vision heavily depends on adequate IT means supporting these tasks, which e.g. manifests in the development of workflow notations and management solutions.

One fundamental problem with existing solutions that has been experienced in practice is that modelling notations like event process chains (EPCs) or the business process modelling notation (BPMN) are activity oriented and thus focus heavily on the ordering and conditions of activity execution. This leads to a particular neglect of the underlying process motivations, which are only implicitly existing during workflow elicitation. Without this so called workflow context perspective [2] it becomes e.g. difficult to optimize the processes because it cannot be easily afforded why specific activities in the workflow exist and if they e.g. could be completely cut out. These observations led to the development of more abstract, flexible and agile workflow concepts. While these concepts contribute to closing the gap between business view and IT view, the greater conceptual abstraction also partially hides the complexities of all possibilities of the exact runtime control flow and reduces the ability of workflow engineers to easily identify specification flaws in the workflow.

In order to equip a workflow modeller with a toolset to better understand the possible runtime behaviour of more abstract workflows, simulation and validation gain importance. In this paper a new concept and implementation for a simulation based validation approach of workflows is presented. It allows to specifiy execution scenarios and automatically evaluate them with respect to expected process outcomes. The approach does not depend on the concrete workflow notation in which processes are described but only assumes that specific workflow management facilities are available. Thus, the approach is viable for validating traditional e.g. BPMN based workflows as well.

The rest of the paper is structured as follows. In Section 2 related work with respect to verification and validation approaches of workflows is discussed. Thereafter, in Section 3 the concept of the new simulation based validation approach is presented. Its implementation and usage is explained in Section 4 and the usefulness of the approach is further illustrated by an example application taken from our industry cooperation partner Daimler AG. The paper concludes with a summary and a short outlook on possible future enhancements.

2 Related Work

Literature and practical application contains a large variety of validation and verification approaches for workflows. They can roughly be divided into two categories. The first category consists of formal static analysis of the workflow model to identify conceptual or implementation flaws. The second category comprises of validation using simulation-based execution of the workflows.

Formal verification approaches of workflows apply a number of techniques, such as propositional logic [3], model verification [4] and graph reduction [5]. The approaches are usually either based on a formally well-defined representation like petri nets [6] or attempt to translate workflows implemented in different representation like EPCs or BPMN into a more approachable form from a formal perspective [7]. While formal verification has the distinct advantage of guaranteeing correctness within the given constraints of the verification approach, translation of workflow models weakens this guarantee unless the translation itself is formally proven correct. Verification also requires a well-defined language with well-known properties, however, some languages used in practices lack these criteria. For example, many parts of BPMN in its current form are left ambiguous and underspecified. The reason for this usually is not negligence but an attempt to bridge the gap between the technical side and the business side of workflows. Parts of the specification are deliberately left fuzzy with ambiguous semantics which enables the use of the specification in informal and non-technical settings. However, some of the issues of ambiguity are currently addressed in the upcoming version 2.0 of the BPMN specification.

Furthermore, verification approaches are often quite limited on what they can guarantee. For example, while there are excellent approaches to guarantee correctness of the workflow diagram like ensuring that branches in the workflow are terminated with a correct join element, they are generally unable to verify non-trivial semantics like task instructions written in a programming language or complex runtime behavior.

This problem can be approached by the validation techniques in the second category. Instead of statically analyzing the workflow models, the workflow is executed in a simulated environment which attempts to imitate the environment in which the workflow is planned to be deployed. Examples of this approach include a variety of tools like LSIM [8], iGrafx Process [9], the Corporate Modeler Suite [10] and the ARIS Toolset [11]. The tools generally target specific workflow notations such as BPMN to combine static analysis described above with simulation. Furthermore, the focus of the tools tend to be performance measurement and optimization. For example, the ARIS Toolset offers a large number of features used to measure operating and performance figures of the workflow during simulation.

In contrast, the focus of our approach has primarily been the validation of the workflow using predefined test cases centering around expected real world scenarios. The disadvantage of this approach compared to verification is that the correctness can only be ensured within the given conditions of the test. Since the number of configurations in any non-trivial workflow is very large, exhaustive search becomes infeasible. This means the approach can only validate the workflow to a certain degree and may not detect all errors present. The advantage of the approach is that the complexity of the workflow language is irrelevant to the test as long as there is a workflow engine capable of executing the workflow. Furthermore, the current approach uses few assumptions about the workflow itself allowing the system to be useful for different kinds of workflow notations. While the focus has been on validation of workflows, the approach can be extended to include simulation of the environment in which the workflow is running but which is, by itself, not part of the workflow. This includes logistic operations, production machinery, behavior of workflow participants and market influences. This allows additional applications beyond workflow validation like workflow optimization similar to the tools mentioned above.

In the following we will present this simulation-based testing approach which uses simulation of workflow participants to validate the correctness of workflows. The focus will be on goal-oriented and agile workflows, however, as mentioned, the approach itself can at least partially be applied to any workflow which relies on interaction with participants.

3 Validation Approach

Due to its application in industry and business automation and intense research interest, a great variety of workflow approaches and notations are available. While our validation approach is designed to be generic, two kinds of notations in particular were the focus of the project. The first is the well-known *Business Process Modeling Notation* (BPMN, see [12]), which has been extended with task and edge annotations, allowing it to be directly executed by an interpreter. While BPMN does not strictly define semantics and contains certain ambiguities, most BPMN elements can be interpreted in a straightforward fashion. Furthermore, the updated standard BPMN 2.0 specifically aims to provide execution semantics. Eventually, the interpreter aims to support these semantics, however, for now it supports a restricted set of BPMN where the semantics appear to be clear and are ultimately defined by the implementation of the interpreter as included in the Jadex platform.

In addition, an additional notation called *Goal-oriented Process Modeling Notation* (GPMN, see [13]) has been developed, which allows the description of goal-oriented workflows using goal hierarchies. Workflows implemented using GPMN are converted into BDI agents at runtime. BDI agents are based on the Belief-Desire-Intention model where beliefs represents the agent's current knowledge about the world, goals represent its abstract desires of what should be accomplished and plans represent concrete intents of the agent with explicit actions the agent follows (see [14]). The goals used in GPMN workflows are directly converted to goals of the resulting agents while the plans that are on the leaf nodes of the goal hierarchy are represented by small workflow fragments implementing the concrete steps in BPMN. During execution, the GPMN-derived BDI agent employs a BPMN interpreter to execute the fragments as plans of the agent.

In addition, GPMN workflows contain a context which represents the current workflow state. This context is used as the belief base of the converted BDI agent at runtime. This context may be changed during execution of the workflow, either directly by the workflow itself or by effects outside the workflow. Changes in the context can directly affect the workflow and thus the agent behavior by influencing adoption, persuit and rejection of goals.

3.1 Goals of the Test System

In practice, workflow engineers currently often validate workflows by manually performing tests of common scenarios which represent typical business cases for the workflow. For example, if the workflow involves production preparation for a new vehicle line at Daimler, the workflow engineer executes the workflow and then plays out the scenario as it is typically expected to occur at Daimler. This involves providing various production issues such as faulty parts. The information is provided to the workflow in the usual fashion as if the workflow is used during production. The behavior of the workflow is then monitored and irregularities are noted and addressed during reengineering.

As a first step to provide better testing facilities for GPMN, we aim to provide a test system which automates the process the workflow engineers are used to and enable them, through automation, to increase test coverage and test a greater number of test cases which would usually be omitted due to time constraints. The general idea is to make workflows executable in a similar way as test cases for a normal programming language. As such a normal test case, also a workflow test case needs to be specified in terms of inputs and expected outcomes. Given these information are available, a test system can automatically executed the worfklow and validate its results. In addition, workflows in the real world may last a considerable time hindering the efficient execution as test case. In order to alleviate this problem, the test system will employ simulation techniques for an execution that is 'as fast as possible', i.e. the processing power and not waiting times in the process determine its execution speed.

The goal of the current system is neither to provide a full formal verification through model checking nor is it to include complete test coverage since the complexity explosion with non-trivial workflows would impede its usefulness in practice. Since the system is part of a system dealing with business process management, it needs to be accessible to people without a technical or formal background such as business users. It allowss business users to test identified domain scenarios in a sensible way and in this helps increasing the test coverage of the workflows. The system is therefore more closely resembling other test techniques for general purpose programming languages such as unit tests. However, it is more specifically focused on aspects of workflow management and the workflow language GPMN.

3.2 Requirements for Automated Testing

Both GPMN and BPMN offer language features which allow the workflow engineer to implement different execution paths or branches. In the case of BPMN workflows, this is accomplished using the gateway element, which can split the control flow into either multiple paths executed concurrently or diverts the flow towards one of multiple possible control flow edges. The implementation of execution paths in GPMN workflows is more subtle and indirect. Depending on the state of the workflow context, different goals may become active resulting in the execution of different BPMN plans. This control flow subtlety of implicit control flow paths in GPMN workflows increases the difficulties of a workflow engineer to accurately predict possible runtime execution paths and is the primary motivation for our validation approach.

The core idea of our approach towards validating such workflows is the use of automated tests. This implies that the system should be able to execute such workflows without user intervention once the test has started. Since only the most trivial workflows can be automatically executed merely using a workflow engine and since most workflows require interaction with workflow participants or automated systems while running, additional system components are necessary beyond the workflow engine itself. Moreover, while most workflows specify a range of possible responses by workflow participants, they generally do not specify which responses will influence workflow behavior, thus necessitating the specification of additional information by the workflow engineer before the test.

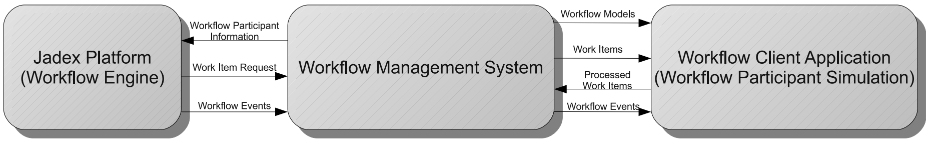

Jadex Platform (Workflow Engine)
Workflow Participant Information
Work Item Request
Workflow Events
Workflow Management System
Workflow Models
Work Items
Processed Work Items
Workflow Events
Workflow Client Application (Workflow Participant Simulation)

Fig. 1. Structure of the proposed test system

Consequently from a simulation perspective, a workflow engine executing a workflow is not a viable simulation model for validation workflows since it lacks sufficient detail even for simple automated execution, much less being a realistic representation of a production environment using workflows. Therefore it is necessary to add additional components to the system which are equivalent or at least sufficiently similar to their production counterparts to represent an adequate model of a workflow in production use (see Figure 1).

3.3 Workflow Management System

One component which is routinely part of workflow systems in businesses is a *workflow management system* (WfMS). The task of a WfMS among other things is to facilitate interaction between workflows and workflow participants. This is generally done by providing work items, which are packages generated by the WfMS on behalf of the workflow containing all the information needed by the workflow participant to accomplish their part of the workflow. The WfMS then distributes the work items among the workflow participants using a variety of approaches such as roles. As a result, the simulation model of a realistic test system needs to include a WfMS which accurately represents a WfMS used in production.

Work items generated by the WfMS not only include information for the workflow participant but often ask the participant to gather and provide external information like customer data or processed documents for the workflow. This is defined in the workflow with the specification of typed parameters in tasks which generate work items. This information often influence further behavior of the workflow at critical junctions like BPMN gateways or goal deliberation. Since real workflow participants are not available during test runs, it is necessary to simulate their actions, including the supply of external information.

Work items are generally retrieved and processed by the workflow participant using a workflow application client interacting with the WfMS. The work items are retrieved, processed by the workflow participant and finally comitted back to the workflow management system, thus allowing the workflow to continue executing. In order to simulate this behavior, the workflow client application used by the workflow participant needs to be replaced with an automated workflow client application which simulates its behavior and the behavior of the workflow participant.

3.4 Client Application

The simulated workflow application client is required to provide the information which is normally provided by the workflow participant. As a first step, the client identifies the workflow tasks which generate work items and require the workflow participant to provide information in the form of work item parameters by examining the workflow model and the models of possible subprocess, such as BPMN workflow fragments in case of GPMN workflows. Parameters are typed and thus already have a limited parameter space. However, this parameter space in cases such as string types is extremely large, precluding an exhaustive test of the full parameter space. Since a complete verification of the process using this approach would also require to test the cartesian product of the parameter space of all parameters provided by the workflow participant, the complexity of such a verification exceeds the limits for a practical test and cannot be considered a useful approach.

As a result, it is necessary to restrict the scope of the test to only include the part of the parameter space which includes the most promising cases, such as corner cases of branching decisions and validating the workflow only for those cases. Since the test cases cannot be identified automatically, the workflow engineer has to define the parameter space that needs to be tested. This is accomplished by the system by allowing the workflow engineer to define test *scenarios*. Scenarios represent a subset of the full parameter space of the workflow participant interaction with the workflow. For each parameter in the process the workflow engineer can define a set of parameter values which are used to process work items while the workflow is executing. If the workflow engineer defines multiple values for each parameter within the same scenario, the cartesian product of those values is tested at runtime.

The workflow engineer can define multiple scenarios for each workflow. When the test is started, the first scenario is selected and the workflow is started repeatedly, once for every element of the cartesian product of the parameter values in that scenario. Once the scenario finishes, the next scenario is selected until all scenarios have been tested. An event log is kept during each execution, recording notable events occuring at runtime for later analysis. The workflow engineer can use this log to identify errors in the workflow and correct them before the workflow is used in a production system. In addition, a test report is generated and can be reviewed.

Errors in the workflow can be unrecoverable states of the workflow like a raised exception during execution or unintended behavior of the workflow such as performing a faulty execution order of tasks. In additions, it is also considered to be an error if the workflow returns the wrong results or reaches the wrong internal state. Since the workflow engineer has to be informed about such errors occuring, monitoring of the workflow is required. On raised exceptions, executed steps of the workflow and state changes the workflow engine can generate a workflow event which is passed through the WfMS to the client application for review by the workflow engineer. It would also be desirable to allow the workflow engineer to define a validation function which receives the information provided

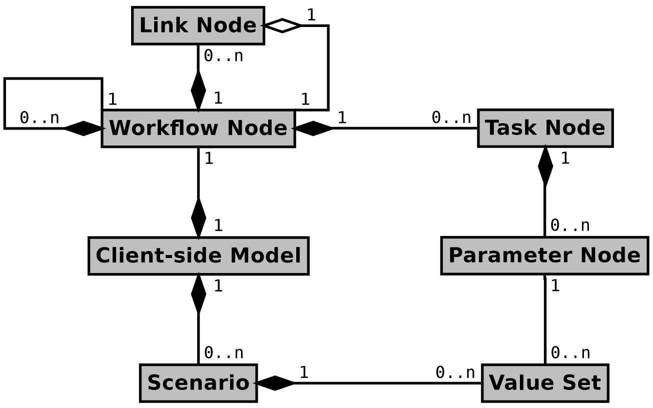


Fig. 2. The workflow tree model used on the client side of the testing system

to the workflow and the final state and result of the workflow allowing the system to automatically evaluate whether a test was successful.

The following section will elaborate on the individual parts of the testing system. It will include an overview of the workflow management system and provide details about the simulated workflow client application and how the workflow engineer can define the parameter space subset for each scenario.

4 Simulation System Components

As mentioned in the previous section, the testing system requires a minimum of three components. In order to execute the workflows themselves, a workflow engine is needed, a workflow management system is needed for work item management and user interaction and finally there needs to be a special workflow application client which simulates user behavior.

Since GPMN workflows are translated into BDI agents with BPMN plans and thus require a workflow engine which can execute both, the Jadex Active Components Platform [15] has been chosen as the execution environment for the workflows. The platform is not only capable of executing GPMN-derived BDI agents but also includes a BPMN interpreter which allows the execution of BPMN processes alongside agents. While the platform is able to execute standalone BPMN processes as active components, BPMN workflow fragments which represent GPMN plans are executed with a special BPMN plan interpreter, which allows the BPMN workflow fragments to access the GPMN context in the form of the agent belief base.

4.1 Workflow Management System Architecture

A primary function of a workflow management system is the generation and distribution of workitems which is triggered by user tasks in workflows. Workitems instruct a workflow participant, such as an employee involved in the process, to perform a certain task for the workflow. The workitem itself does not always specify the exact person performing the task in the workflow model since such a specification is not always necessary if the task can be performed by multiple people. In fact, such a direct assignment of tasks can be problematic due to dynamically changing conditions with regard to the status of persons, such as vacations and employee fluctuations. Therefore the assignment is usually done at runtime using specific criteria. A common way of solving the problem is to assign the task to a specific role in the workflow model and letting either the WfMS or the employees themselves do the final assignment to a specific person depending the person being able to fulfill that role.

Since workitem assignment is part of the runtime behavior of workflows, this aspect alone requires the inclusion of the WfMS in the simulation model to accurately reflect that behavior during simulation. Additionally, there are further WfMS runtime behavior that should be included in the simulation model, such as workitem access rules and workflow instantiation.

The implemented WfMS is implemented largely based on the reference model of the Workflow Management Coalition [16]. It uses the Jadex platform with Jadex platform services implementing the services required for the system which will be explained in further detail in this section. In addition, the system includes three interface agents which realize a message-based interface with workflow application clients.

The services of the WfMS are divided into internal and external services, the former implementing the actual functionality of the WfMS while the latter, represented by the agents, act as an interface which can be used by workflow client applications to connect with the WfMS. The Jadex platform itself represents the workflow engine and enactment service of the WfMS by providing the necessary support for instantiation of workflow models.

Workflow models are managed by the WfMS using a process model repository service. This service supports the addition and removal of process models by employing the Jadex library service which allows Jadex to dynamically load new models, resources and executable code by linking directories or jar-archives. In addition, it offers access to workflow models for workflow client software, which is a necessity for the testing system since the simulated workflow client application requires the workflow model in order to identify tasks in the workflow which require interaction with a workflow participant.

The Authentication, Access Control and Accounting (AAA) Service of the WfMS provides additional workflow participant-centric services. This includes access control to workflow services and a role management system which allows the WfMS to associate work items with workflow participants. Each task which generates work items can assign a role to the work item, restricting this work item to workflow participants who represent that role. Work items without a designated role become available to any workflow participant connected to the system.

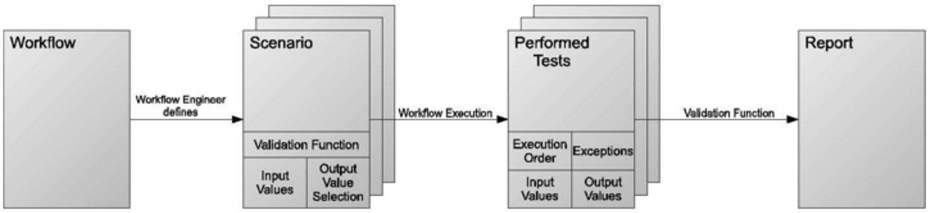


Fig. 3. Procedure for testing workflows

The external services of the WfMS consist of three parts. The first service is the *workflow client interface*, which manages the work item queue and distributes work items to connected clients. The second service is the *process definition interface*, which provides access to the process model repository by allowing clients to add and remove workflow models and accepting requests for workflow models. Finally, the *administration and monitoring service* offers access Vto administrative and monitoring functions. The monitoring functions are especially critical for the testing system since they provide feedback regarding events happening during workflow execution.

This system provides a similar functionality to a workflow system in production use. In addition to this basic system, a workflow application client which simulates the behavior of workflow participants is needed to create an automated testing system which can execute tests of workflows without user intervention. This client is implemented as a BDI agent which connects to the three external service agents to the WfMS. The agent provides a user interface to the workflow engineer, which allows them to open the desired workflow model which is retrieved from the WfMS.

As a result the WfMS forms an integral part of the simulation model by simulating the WfMS used in a production system. This is especially important if a specific workitem assignment based on roles or similar concepts is part of the validation parameters for the workflow. In addition, potential effects of workitem processing such as multiple available workflow, workitem access rules and communication delays can be tested.

4.2 Client-Side Workflow Model

Since it is desirable for the workflow engineer to gain an overview of the parts of the workflow which are relevant to interaction with workflow participants, the client agent uses the workflow model to generate a tree representation of the workflow model which can be seen in Figure 2. The tree consists of a *workflow node* as the root node, which represents the workflow originally opened by the workflow engineer. If a workflow contains sub-workflows like BPMN plans, the children of its workflow node can include further workflow nodes representing those sub-workflows.

The sub-workflow graph of a workflow can contain cycles, for example, when a sub-workflow uses its parent as a sub-workflow. This would suggest a graph representation to be the natural form for representing the workflow structure. However, cycles in the workflow graph are rare in practice and a workflow engineer would expect a tree form rather than a graph. Therefore the tree representation is more desireable, nevertheless the special case of a cyclic workflow structure should be supported. This problem is solved with the use of *link nodes*. If a particular sub-workflow is found again after having been found before, it is represented as a link node in the tree. This link node is a simple reference to the first occurance of the sub-workflow in the structure and no further expansion of the tree is done beyond this reference to avoid endless expansion.

If a workflow node is a BPMN workflow or workflow fragment its child nodes can, in addition to sub-workflow nodes, contain *task nodes* which represent tasks containing interaction with workflow participants. Lastly, the children of task nodes are *parameter nodes*, which represent typed parameters which would ordinarily be provided by a workflow participant.

4.3 Scenarios

The tree structure of a modelled workflow is presented to the workflow engineer in graphical form in the user interface. This allows them to define *scenarios*. Scenarios consist of sets of input values for each parameter of the workflow, definition of data collected from the workflow during the simulation and a validation function which is used to evaluate the success of the test and generate a report for the workflow engineer. Figure 3 demonstrates the use of scenarios in the full test procedure.

For each of the input parameters, the workflow engineer can add multiple values depending on the type of the parameter. The cartesian product of the input values in the scenario is used to create tests for the workflow which means that a minimum of a single value for each parameter is required for the scenario to generate at least a single viable test.

The workflow engineer can also select what is monitored during execution. This can include output values of the workflow, its final internal state, exceptions and the task executed. These values are passed to the *validation function* after execution to determine whether the test was a success and to generate a useful report. The validation function is also defined by the workflow engineer and included in the scenario.

Multiple scenarios can be defined for each workflow which can be automatically executed in succession. The total number of required test runs of the workflow are reported to the workflow engineer while assembling the scenarios. Since the cartesian product of multiple parameter values in a scenario quickly increases the complexity of the whole test, the workflow engineer has to carefully chose the tests and carefully balance between adding additional scenarios for the test run or adding additional parameter values to a single scenario.

The results of a test run can be reviewed in the report generated by the validation functions of the scenarios and is displayed to the workflow engineer in the client application. In addition, several tools provided by the Jadex platform can be used during the simulation as well. This includes an introspector tool for

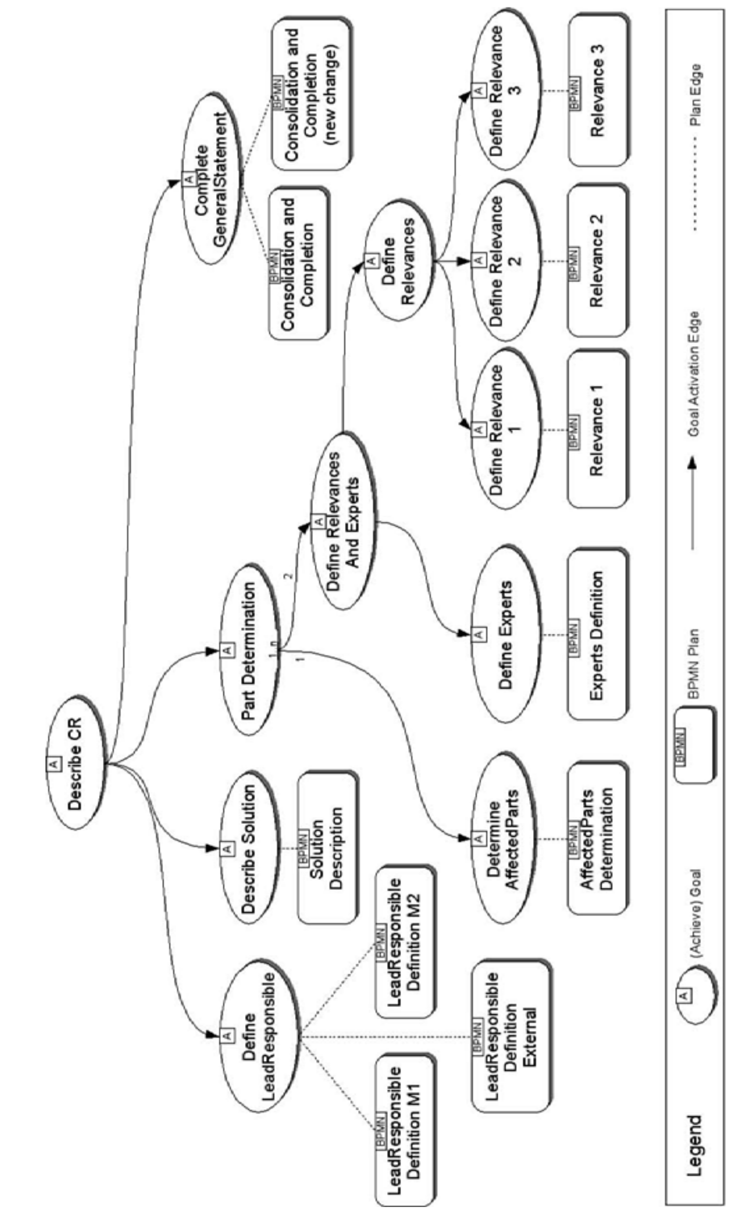


Fig. 4. The goal hierarchy of the Active Change Management workflow

investigated the state of the workflow and a message center tool for monitoring messages between workflows and their support systems like the WfMS and the client application.

The next section will present an example use case for an industrial workflow used for change management as envisioned by Daimler AG and will demonstrate how the test system can be used to find implementation errors in advance of deployment.

5 Example Use Case

This section will present how the system can be used to validate a workflow. The workflow was developed by Daimler Group Research and represents an industrial workflow used for change management (see [17]). Change management workflow coordinate the process of developing and implementing changes for an existing product and ensures that production line changes and adaptions of the physical geometry of the product are performed in order to allow a smooth introduction of the changed product.

Since the workflow is very large and complex, this section will focus on a smaller subset of this workflow (cf. Figure 4). This subset involves the gathering of information about the planned change to the product, designating key personnel and assigning required resources. The final result of this part of the process is a description of the change request and requirements which will be used in the later part of the workflow to perform the change.

The process fragment contains a single goal for defining the change request. This goal is decomposed into subgoals, some of which contain context conditions which suspend their execution until the context has the required state for the goal to be adopted. Goals without further subgoals are associated with plans which are implmented by BPMN workflow fragments.

After implementation, one of the BPMN workflow fragments contained an error. The BPMN fragment containing the error was the fragment determining the parts affected by the change in the product. During the execution of the fragment, the leading developer responsible for the change request is required to enter the parts of the product which are affected by the change. The developer has the choice to do this using three different ways of providing this list of parts. The first way is to provide a list of serial numbers of the affected parts. The second way is to provide a drawing which is processed for part information and finally, the developer can give a structured description of the components affected by the change.

The first task lets the developer choose between those three ways of providing the part list (cf. Figure 5). The first task in this fragment generates a work item containing a list of three strings which represent the choices of the developer. The developer can select one of the strings and commits the work item. The string is then passed to the gateway, and compared to strings provided by the edges behind the gateway branch. If the string matches, the process continues executing using that path and provides the developer with a new work item which contains the information for the chosen method of entry.

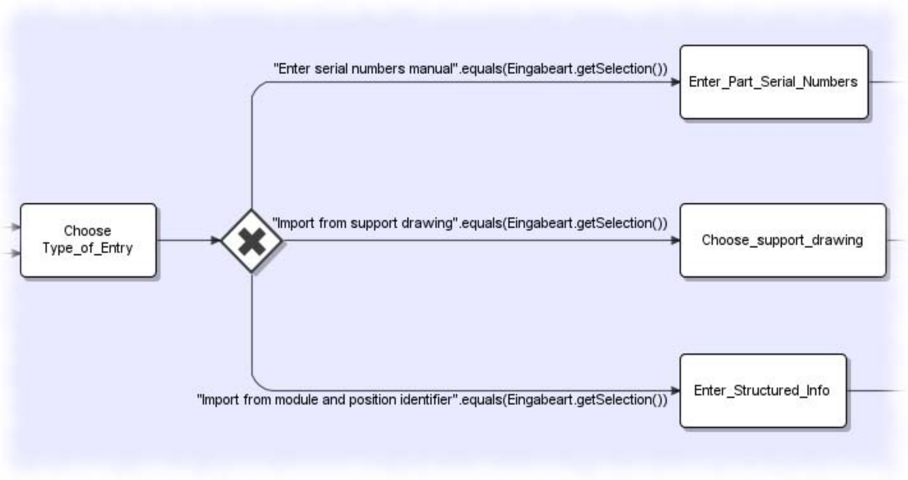
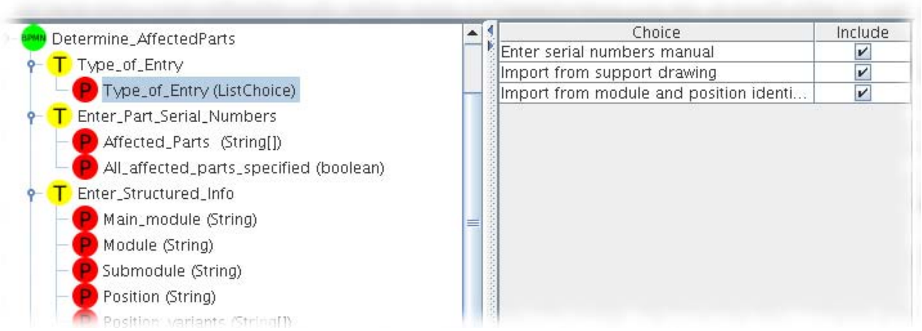

"Enter serial numbers manual".equals(Eingabeart.getSelection())
Enter_Part_Serial_Numbers
Choose Type_of_Entry
"Import from support drawing".equals(Eingabeart.getSelection())
Choose_support_drawing
"Import from module and position identifier".equals(Eingabeart.getSelection())
Enter_Structured_Info

Fig. 5. The leading developer of the change has three ways of providing a list of affected parts

The implementation error in this part of the process was that one of the strings provided by the edges did not match with the corresponding string generated by the entry type selection task. Since none of the edges on the gateway branch are marked as default edge which would be taken if no other edge matches, the workflow will terminate with an exception if the developer selects the faulty choice.

This error was found using the test system. A scenario had been created test each of the branches leaving the gateway in this workflow fragment. For every parameter in the workflow except the entry method choice of the developer a single value was added to the scenario. All three possible strings were then added to the parameter concerning the entry choice resulting in a scenario which


Determine_AffectedParts
Type_of_Entry
Type_of_Entry (ListChoice)
Enter_Part_Serial_Numbers
Affected_Parts (String[])
All_affected_parts_specified (boolean)
Enter_Structured_Info
Main_module (String)
Module (String)
Submodule (String)
Position (String)
Position_variants (String[])

Choice	Include
Enter serial numbers manual	✓
Import from support drawing	✓
Import from module and position identi...	✓

Fig. 6. The scenario for testing the entry choice branch contains all three possible parameter values used

specifically target this branch in the workflow (cf. Figure 6). In addition, example entries for the part specification had been added in order to verify the correct function of the workflow in identifying the parts affected by the change, test corner case entry such as empty strings for parts and test another branch in the workflow where the developer has to confirm the list of parts or otherwise cause a restart of the workflow fragment. The test complexity of the scenario required 972 test runs which were executed on an Intel i5 CPU clocked at 2.67GHz in less than a minute.

During one test run, the faulty string was selected as part of the scenario. This resulted in an exception and the termination of the workflow. This event was noted in the simulation log, including the parameter configuration used when the error occured. This result allowed the workflow engineer to locate the fault in the workflow and correct the problem.

6 Summary and Future Enhancements

This paper has presented a simulation-based validation approach for workflows. The approach allows specifying execution scenarios in form of test cases, which include ranges of input values to be tested and defined output states to be reached for a successful execution. The approach is especially well-suited for agile process descriptions with abstract specification means, such that possible process execution paths cannot easily be predicted. Nevertheless, the approach also works well with traditional process specification languages like BPMN. The implementation of the validation approach is based on the process execution facilities of the Jadex active component platform and provides additional tools for the specification, execution and validation of scenarios. The applicability of the approach has been exemplified by a case study from our project partner Daimler AG. It has been shown how the validation approach allows testing a modeled process to find and resolve existing problems in the process description.

Two interesting areas for future work are envisaged. First, the approach can be extended towards being used not only for process validation, but also for process analysis and optimization. Extending the simulation engine with elaborated analysis tools would allow measuring the quality of processes and benchmarking alternative processes against each other. Second, instead of using pre-specified test cases as scenarios, complex simulation models could be used to dynamically produce realistic test data. E.g. for logistics management processes, a simulation model of a supply chain could be connected to the process engine and provide input data for the workflow application to be tested.

In addition, while initial feedback from our industry partner has been positive, a full evaluation of the system is desirable. However, the test system is part of a larger package of software tools which aim to provide a full Business Process Management Suite, eventually supporting a full BPM lifecycle. The scope of the evaluation should therefore include the full lifecycle of which the test system is an integrated part. However, a number of components of the suite are still in development. Once these components have reached a sufficient state, a full evaluation of the suite, including the test system, will be conducted to assess the usefulness of the system in practice.

Acknowledgement. We would like to thank the DFG for supporting the technology transfer project Go4Flex.

References

1. Schmelzer, W.S.H.J.: Geschäftsprozessmanagement in der Praxis. Hanser Fachbuchverlag (2008)
2. List, B., Korherr, B.: An evaluation of conceptual business process modelling languages. In: SAC 2006: Proceedings of the 2006 ACM symposium on Applied computing, pp. 1532–1539. ACM, New York (2006)
3. Bi, H.H., Zhao, J.L.: Applying propositional logic to workflow verification. Information & Software Technology 5, 293–318 (2004)
4. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based verification of web service compositions. In: 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada (2003)
5. Sadiq, W., Orlowska, M.E.: Analyzing process models using graph reduction techniques. In: The 11th International Conference on Advanced Information System Engineering., vol. 25(2), pp. 117–134 (2000)
6. van der Aalst, W.M.P.: Workflow verification: Finding control-flow errors using petri-net-based techniques. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 161–183. Springer, Heidelberg (2000)
7. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in bpmn. Information & Software Technology 50(12), 1281–1294 (2008)
8. Enstone, L.J., Clark, M.F.: BPMN and Simulation, Lanner Group Limited (2006), `http://www.dynamic.co.kr/Witness_Training_Center/Articles/Bpmn%20-%%20simulation.pdf`
9. iGrafx Process Corel Inc. (2009), `http://www.igrafx.de/products/process/index.html`
10. Corporate Modeler Suite Casewise Ltd. (2009), `http://www.casewise.com/Products/CorporateModelerSuite/`
11. Scheer, A.-W., Nüttgens, M.: ARIS architecture and reference models for business process management. In: van der Aalst, W.M.P., Desel, J., Oberweis, A. (eds.) Business Process Management. LNCS, vol. 1806, pp. 376–509. Springer, Heidelberg (2000)
12. Business Process Modeling Notation (BPMN) Specification, Object Management Group (OMG) (February 2008), `http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf`
13. Braubach, L., Pokahr, A., Jander, K., Lamersdorf, W.: Go4Flex: Goal-oriented process modelling. In: Essaaidi, M., Malgeri, M., Badica, C. (eds.) Intelligent Distributed Computing IV. SCI, vol. 315, pp. 77–87. Springer, Heidelberg (2010)
14. Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)

15. Pokahr, A., Braubach, L., Jander, K.: Unifying agent and component concepts. In: Dix, J., Witteveen, C. (eds.) MATES 2010. LNCS, vol. 6251, pp. 100–112. Springer, Heidelberg (2010)
16. Workflow Reference Model, Workflow Management Coalition (WfMC) (January 1995), http://www.wfmc.org/reference-model.html
17. Burmeister, B., Arnold, M., Copaciu, F., Rimassa, G.: Bdi-agents for agile goal-oriented business processes. In: AAMAS 2008: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, pp. 37–44 (2008)

Augmenting Agent Platforms to Facilitate Conversation Reasoning

David Lillis and Rem W. Collier

School of Computer Science and Informatics
University College Dublin
{david.lillis,rem.collier}@ucd.ie

Abstract. Within Multi Agent Systems, communication by means of Agent Communication Languages (ACLs) has a key role to play in the co-operation, co-ordination and knowledge-sharing between agents. Despite this, complex reasoning about agent messaging, and specifically about conversations between agents, tends not to have widespread support amongst general-purpose agent programming languages.

ACRE (Agent Communication Reasoning Engine) aims to complement the existing logical reasoning capabilities of agent programming languages with the capability of reasoning about complex interaction protocols in order to facilitate conversations between agents. This paper outlines the aims of the ACRE project and gives details of the functioning of a prototype implementation within the Agent Factory multi agent framework.

1 Introduction

Communication is a vital part of a Multi Agent System (MAS). Agents make use of communication in order to aid mutual cooperation towards the achievement of their individual or shared objectives. The sharing of knowledge, objectives and ideas amongst agents is facilitated by the use of Agent Communication Languages (ACLs). The importance of ACLs is reflected by the widespread support for them in agent programming languages and toolkits, many of which have ACL support built-in as core features.

In many MASs, communication takes place by way of individual messages without formal links between them. An alternative approach is to group related messages into conversations: "task-oriented, shared sequences of messages that they observe, in order to accomplish specific tasks, such as a negotiation or an auction" [1].

This paper presents the Agent Conversation Reasoning Engine (ACRE). The principal aim of the ACRE project is to integrate interaction protocols into the core of existing agent programming languages. This is done by augmenting their existing reasoning capabilities and support for inter-agent communication by adding the ability to track and reason about conversations. Currently at the stage of an initial prototype, ACRE has been integrated with several agent programming languages running as part of the Common Language Framework of the Agent Factory platform [2]. The longer-term goals of ACRE include its use within other mainstream agent frameworks and languages.

The principal aim of this paper is to outline the goals of the ACRE project and to discuss its integration into Agent Factory.

This paper is laid out as follows: Section 2 outlines some related work on agent interaction. Section 3 then provides an overview of the aims and scope of the ACRE project. The model used to reason about conversations is presented in Section 4. ACRE protocols are defined in an XML format that is outlined in Section 5, followed by an example of a conversation in execution in Section 6. Details of the integration of ACRE into the Agent Factory framework are given in Section 7. Finally, Section 8 outlines some conclusions along with ideas for future work.

2 Related Work

In the context of Agent Communication Languages, two standards have found widespread adoption. The first widely-adopted format for agent communication was the Knowledge Query and Manipulation Language (KQML) [3]. An alternative agent communication standard was later developed by the Foundation for Intelligent Physical Agents (FIPA). FIPA ACL utilises what it considers to be a minimal set of English verbs that are necessary for agent communication. These are used to define a set of performatives that can be used in ACL messages [4]. These performatives, along with their associated semantics, are defined in [5].

Recognising that one-off messages are limited in their power to be used in more complex interactions, FIPA also defined a set of interaction protocols [6]. These are designed to cover a set of common interactions such as one agent requesting information from another, an agent informing others of some event and auction protocols.

Support for either KQML or FIPA ACL communication is frequently included as a core feature in many agent toolkits and frameworks, native support for interaction protocols is less common. The JADE toolkit provides specific implementations of a number of the FIPA interaction protocols [7]. It also provides a Finite State Machine (FSM) behaviour to allow interaction protocols to be defined. Jason includes native support for communicative acts, but does not provide specific tools for the development of agent conversations using interaction protocols. This is left to the agent programmer [8, p. 130]. A similar level of support is present within the Agent Factory framework [9].

There do exist a number of toolkits, however, that do include support for conversations. For example, the COOrdination Language (COOL) uses FSMs to represent conversations [10]. Here, a conversation is always in some state, with messages causing transitions between conversation states. Jackal [11] and KaOS [12] are other examples of agent systems making use of FSMs to model communications amongst agents. Alternative representations of Interaction Protocols include Coloured Petri Nets [13] and Dooley Graphs [14].

3 ACRE Overview

ACRE is aimed at providing a comprehensive system for modelling, managing and reasoning about complex interactions using protocols and conversations. Here, we distinguish between *protocols* and *conversations*. A protocol is defined as a set of rules that dictate the format and ordering of messages that should be passed between agents that are involved in prolonged communication (beyond the passing of a single message). A conversation is defined as a single instance of multiple agents following a protocol in order to engage in communication. It is possible for two agents to engage in multiple conversations that follow the same protocol.

Such an aim can only be realised effectively if a number of features are already available. These include:

– **Protocol definitions understandable by agents:** Interaction protocols must be declared in a language that all agents must be able to understand and share. This also has the advantage that the protocol definition is separated from its implementation in the agent, thus providing a programmer with a greater understanding of the format the communication is expected to take. ACRE uses an XML representation of a finite state machine for this purpose. This representation is further discussed in Section 5. The separation of protocol definitions from agent behaviours also facilitates the development of external tools to monitor communication between agents.
– **Shared ontologies:** A shared vocabulary is essential to agents understanding each other's communications. A shared ontology defines concepts about which agents need to be capable of reasoning.
– **Plan repository:** With the two above features in place, an agent may reason about the sequence of messages being exchanged, as well as the content of those messages. This reasoning will typically result in an agent deciding to perform some action as a consequence of receiving certain communications. In this case, it is useful to have available a shareable repository of plans that agents may perform so that new capabilities may be learned from others. Clearly, the use of shared plans will be dependent on the agent programming language(s) being used.

The presence of these features aid greatly in the realisation of ACRE's aims. The principal aims are as follows:

– **External Monitoring of Interaction Protocols:** At its simplest level, conversation matching and recognition of interaction protocols allows for a relatively simple tool operating externally to any of the agents. This can intercept and read messages at the middleware level and is suitable for an open MAS in which agents communicate via FIPA ACL. This is a useful tool for debugging purposes, allowing developers to monitor communication to ensure that agents are following protocols correctly. This is particularly important where conversation management has been implemented in an ad-hoc way, with incoming and outgoing messages being treated independently and without a strong notion of conversations. In this case, the protocol definitions can be formalised after the implementation of the agents without interfering with the agent code itself until errors are identified.

– **Internal Conversation Reasoning:** On receipt of a FIPA ACL message, it should be possible for an agent to identify the protocol being followed by means of the `protocol` parameter defined in the message (for the specification of the parameters available in a FIPA ACL message see [15]). Similarly, the initiator of a conversation should also set the `conversation-id` parameter, which is a unique identifier for a conversation. By referring to the the protocol identifier, an agent can make decisions about its response by consulting the protocol specification. Similarly, the conversation identifier may be matched against the stored history of ongoing conversations.

  ACRE provides an agent with access to information about the conversations to which it is a party. This allows the agent to reason about this according to the capabilities of the agent programming language being used. One example of this is the use of this information to analyse the status of conversations and generate appropriate goals for the agent to successfully continue the conversation along the appropriate lines for the protocol that is specified. This has previously been done with the AFAPL2 agent programming language [16], following the use of goals in [17]. Goals represent the motivations of the participants in a conversation. Thus the agents' engagement in a particular conversation is decoupled from the individual messages that are being exchanged, allowing greater flexibility in reasoning about their reactions and responses.

– **Organisation of Incoming Messages:** It is possible that an agent communicating with agents in another system may receive messages that do not specify their protocol and/or conversation identifier. In this case, it is useful for the agent to have access to definitions of the protocols in which it is capable of engaging so as to match these with incoming messages so as to categorise the messages.

  In this situation, message fields such as the sender, receiver, message content and performative can be compared against currently active conversations to ascertain if it matches the expectations of the next step of the underlying protocol.

– **Agent Code Verification:** The ultimate aim of ACRE is to facilitate the verification of certain aspects of agent code. In particular, given integration of conversation reasoning into a programming language, it should be possible to verify whether or not an agent is capable of engaging in a conversation following a particular protocol.

4 Conversation Management

ACRE models protocols as FSMs, with the transitions between states triggered by the exchange of messages between agents participating in the conversation. Messages, protocols and conversations are represented by tuples. As a FSM, each protocol is made up of states and transitions, which are also represented by tuples. This section presents these representations and also provides an informal description of the conversation management algorithm used within ACRE.

A *message* is represented by the tuple (s, r, c, ϕ, p, x), where s is the agent identifier of the message's sender, r is the agent identifier of the recipient, c is the conversation identifier, ϕ identifies the protocol, p is the message performative and x is the message content.

Each *protocol* is represented by a tuple (ϕ, S, T, i, F) where ϕ is the protocol's unique identifier, S and T are sets of states and transitions respectively, i is the name of the initial state and F is a set of names of final (terminal) states.

Within these conversations, each *state* is represented by the tuple (n, s, ϕ) where n is the name of the state, s is the status of the state (whether it is a start, end or intermediate state) and ϕ is the identifier of the protocol it belongs to. A *transition* is represented by $(\sigma, \epsilon, s, r, p, x)$. Here, σ and ϵ are the names of the start and end states respectively, s and r are the agent identifiers of the sending and receiving agents respectively, p is the performative of the message triggering the transition and x is the message content.

As a FSM, a protocol can easily visualised as shown in Figure 1. This figure shows a FSM for a simple, one-shot Vickrey-style auction. It shows the states and transitions associated with this protocol. Transitions are triggered by comparison with messages exchanged between the participating agents.

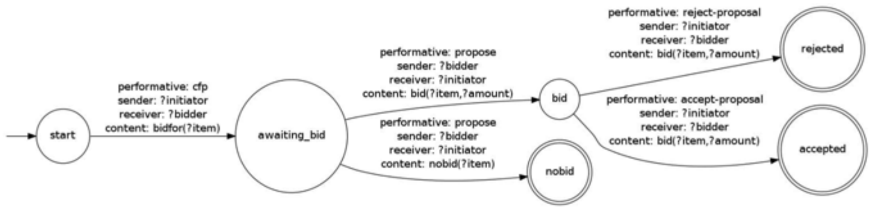


Fig. 1. FSM representation of the Vickrey Auction protocol

Finally, a *conversation* may be represented by (ϕ, A, s, c, B, ψ) where ϕ is the protocol identifier, A is the set of participating agents, s is the name of the conversation's current state, c is the conversation identifier, B is the current set of variable/value bindings and ψ is the conversation status (active, completed or failed).

The values permitted in the tuples shown here are based on first-order logic, meaning that all values are constants, variables or functions. When considering whether a message is capable of advancing a conversation, its fields are compared with the corresponding elements of the conversation's available transitions.

When comparing values, the following rules apply:

– Constant values match against other identical constant values (e.g. in Figure 1, the first transition can only be triggered by a message with the performative `cfp`).

– Variables match against any value.
– Functions match other functions that have the same functor, have the same number of arguments and whose arguments in turn match.

In the pseudocode that follows in Figures 2, 3 and 4, this is encapsulated by the function `matches(a,b)`.

The bindings associated with the conversation (B) is a set of key/value pairs that binds variables to constants or functions against which that they have been matched in triggering a transition. Any variables that have been matched against a constant or function in a triggering message are given a binding that is stored in B. In the example from Figure 1, the sender of the initial message will have their agent identifier bound to the `?initiator` variable, so any further messages must be sent by/to that same agent, whenever the `?initiator` variable is used. This is an example of a variable being used in *immutable context*. Once the variable has been bound to a value, that value may not change for the duration of the conversation.

An alternative approach is to use a variable in a *mutable context*. In this situation, a variable may acquire a binding to a new value regardless of whether it has been previously bound. Further explanation (and examples) of the different variable contexts is presented in Section 5.2. One special-case variable also exists. The *anonymous variable* (denoted by "?") may not acquire any binding. Thus it acts as a wildcard match that will match against any values.

The following sections outline the three key stages of the conversation management algorithm. By convention, elements of tuples are denoted by using subscripts (e.g. the initial state (i) of a protocol (p) is shown as p_i).

4.1 Identifying Candidate Conversations

The first stage of the conversation management algorithm is carried out whenever a message is exchanged and is shown in Figure 2. This identifies any active conversations that may be advanced by a message that has been exchanged. If the message contains a defined conversation identifier (which are unique), then only a conversation bearing that identifier may be advanced by the message. In the event that a message is exchanged without a conversation identifier being present, any conversation with an available transition that may be triggered by the message will be considered a candidate.

A conversation can be advanced by a message if the elements of the message match against the corresponding elements of any available transitions (i.e. that begin at the current state of the conversation). If the message contains a defined conversation identifier, but that conversation cannot be advanced by the message, the status of the conversation must be changed to *failed*.

The `apply(B,a)` function is used to apply a set of bindings (`B`) to a term (`a`). If `a` is a variable used in an immutable context for which a binding exists in `B`, then the bound value is returned. Otherwise, `a` is returned unaltered.

```
$C \leftarrow \emptyset$ to store candidate conversations
$m \leftarrow$ message sent/received
**for each** active conversation $(c)$ **do**
  **if** $m_c = c_c$ **or** $m_c = \perp$ **then**
    **for each** transition $(t)$ **where** $t_\sigma = c_s$ **do**
      **if** $matches(m_s, apply(c_B, t_s))$ **and** $matches(m_r, apply(c_B, t_r))$
         **and** $matches(m_x, apply(c_B, t_x))$ **and** $matches(m_p, t_p)$ **then**
        Add $c$ to $C$
      **end if**
    **end for**
  **end if**
  **if** $m_c = c_c$ **and** $c \notin C$ **then**
    $c_\psi \leftarrow failed$
  **end if**
**end for**
```

Fig. 2. Identifying candidate conversations

4.2 Identifying Candidates for New Conversations

If no active conversations may be advanced by the given message, the second stage is to identify whether the message is capable of initiating a conversation using a known protocol. This procedure is shown in Figure 3.

```
**if** $|C| = 0$ **then**
  **for each** protocol $(p)$ **do**
    **if** $m_\phi = p_\phi$ **or** $m_\phi = \perp$ **then**
      **for each** transition $(t)$ where $t_\sigma = p_i$ **do**
        **if** $matches(m_s, t_s)$ **and** $matches(m_r, t_r)$ **and** $matches(m_x, t_x)$ **then**
          **if** $m_c = \perp$ **then**
            Add $(p_\phi, \{m_s, m_r\}, p_i, nextid(), \emptyset, active)$ to $C$
          **else**
            Add $(p_\phi, \{m_s, m_r\}, p_i, m_c, \emptyset, active)$ to $C$
          **end if**
        **end if**
      **end for**
    **end if**
  **end for**
**end if**
```

Fig. 3. Identifying candidate protocols for new conversations

If the message contains a protocol identifier, then only the protocol with that identifier is considered. Otherwise, the message is compared against the initial transition of each available protocol. On finding a suitable protocol, a new conversation is created and added to the set of candidate conversations (`C`). If the message contained a conversation identifier, this is used as the identifier for the new conversation. Otherwise, a new unique conversation identifier is generated (by means of the `nextid()` function).

4.3 Advancing the Conversation

Having identified conversations that match against the given message, the system must advance a conversation, as appropriate. This is shown in Figure 4. At this stage, events are raised to the agent layer to inform the agent of the outcome of the process. If the message was not capable of advancing or initiating any conversation, an "unmatched" event is raised. If there were multiple candidate conversations (which cannot be the case if conversation identifiers are defined for all messages), an "ambiguous" event is raised.

If one candidate conversation was identified, this is advanced to the next appropriate state. Its bindings must be updated (using the `getBindings(m,t)` function) to include bindings for variables in the transition that were matched against values in the message. The anonymous variable may not acquire a binding. This function does not discriminate between variables based on the context in which they are used. Both mutable and unbound immutable variables are free to acquire new bindings. If an immutable context variable has previously been bound to a value, it is this value that is used in matching the message to the transition (by means of the `apply(B,a)` function shown in Figure 2). As the message content is frequently a function of first order logic, any variables within that function that match against corresponding parts of the message content will also acquire bindings in the same way as standalone variables.

```
**if** $|C| = 1$ **then**
  $c \leftarrow$ the matched conversation in $C$
  $t \leftarrow$ the transition matched by the message $m$
  $c_s \leftarrow t_\epsilon$
  $c_B \leftarrow c_B \cup getBindings(m, t)$
  **if** $c_s$ is an end state **then**
    $c_\psi \leftarrow completed$
    $raiseEvent(completed, c)$
  **else**
    $raiseEvent(advanced, c)$
  **end if**
**else if** $|C| = 0$ **then**
  $raiseEvent(unmatched, m)$
**else**
  $raiseEvent(ambiguous, m)$
**end if**
```

Fig. 4. Advancing the conversation

5 The ACRE XML Format

In ACRE, interaction protocols are modelled using an XML file that follows the ACRE XML protocol schema definition [1]. A sample of an XML representation

[1] `http://acre.lill.is/protocol.xsd`

of a Vickrey Auction Interaction Protocol is given in Figure 5 (this is the same protocol as the FSM in Figure 1). Each protocol is identified by a name, a name and a version number (contained in the `<namespace>`, `<name>` and `<version>` tags respectively). The version number is intended to prevent multiple agents attempting to communicate using different protocol implementations (e.g. if an error is discovered in an earlier attempt at modelling a particular protocol). The use of a namespace helps to avoid conflicts whereby various developers implement different models of similar protocols using the same name.

Each protocol is represented by a number of states and transitions, defined using `<state>` and `<transition>` tags respectively. Each state has only a "name" attribute, so that it can be referred to in the transitions. The type of state each represents (i.e. terminal, initial or other) can be found on the fly when the protocol is loaded. A state at which no transition ends is considered a start state. States at which no transitions begin are terminal states. The reason these are not expressly marked in the protocol definition is because of the ability to import other protocols, which is discussed in Section 5.1.

Transitions are more complex, as these are required to match messages so as to trigger a change in the state of a conversation. Each `<transition>` tag contains up to six attributes, many of which attempt to match against one field of a FIPA message. The attributes in this file correspond with the values in the tuple representing a transition in Section 4. In addition to these message fields, the ACRE conversation manager will also examine the `protocol-id` and `conversation-id` fields to match messages to particular conversations.

The attributes allowable in a `<transition>` tag are as follows:

- **Performative:** This is a mandatory field that specifies the performative that a message must have in order to trigger this transition. The attribute value must be exactly equal to the performative contained in the message for this transition to be triggered. Variables are not permitted in this field.
- **From State:** Another mandatory field, this indicates the state from which this transition may be triggered. If the conversation is in another state then this rule cannot match. In the majority of cases, the attribute value must be the same as the name of a state that is contained either in the protocol itself or in an imported protocol.

  In addition to exact state names, regular expression matching is also permitted. If a regular expression is provided (indicated by beginning and ending the value with a forward slash), then this transition will be triggerable from any state that matches this regular expression. In practice, the protocol interpreter will duplicate this transition for each state name that matches, thus fitting with the model outlined in Section 4.
- **To State:** This is another mandatory field and is used to indicate the state that the conversation will be in upon successful triggering of this transition. As with the "From State" attribute, this should match the name of a state that is either part of the protocol or is imported. However, it may not contain a regular expression, as the conversation state after the sending of a message must be clearly defined.

- **Sender:** This indicates which agent should be the sender of the message. Although it is allowable to use a constant value for this attribute, this is unusual as it specifically restricts the protocol to an agent with a a particular identifier. Generally, this will use a variable to refer to particular agents. The same variable may be used throughout the protocol to indicate that particular messages should be sent by the same agent, as it will have acquired a bound value the first time it matches against an agent identifier.
- **Receiver:** This attribute functions in a similar way to "Sender", with the exception that it is the recipient of the message that is being matched.
- **Content:** This attribute relates to the actual content of the message. It may be a constant, a variable or a function that possibly combines the two. Figure 5 illustrates the use of a function in the `content` field in each of the transitions.

The "Sender", "Receiver" and "Content" attributes are optional in a protocol definition. In each case, the default value if one is not supplied is the anonymous "`?`" variable that matches any value.

5.1 Importing Protocols

One other feature of the ACRE XML format is the ability to import from other protocols. When this occurs, all of the states and transitions from the imported protocol are added to those of the protocol containing the `<import>` tag. This means that transitions in a protocol may refer to states that are not in the protocol itself but rather are in the imported protocol.

One example of a use for this is the "Cancel" meta-protocol that is included in all of the standard FIPA Interaction Protocols. This protocol always works in an identical way, regardless of what the main protocol being followed is: at any non-terminal stage of the conversation, the initiator of the original conversation may terminate the interaction by means of a `cancel` message. This meta-protocol can be extracted into a separate ACRE protocol that is imported by all other protocols that support it.

5.2 Variable Bindings

As mentioned in Section 4, the definition of protocols in ACRE allows the use of three types of variable. The *anonymous variable* "`?`" is an unnamed variable that is capable of matching against any value. As such, it can be considered to be a wildcard match. A transition whose `content` attribute is set to "`?`" can be triggered by a message with any content (assuming the other fields in the message match the specified transition).

Two types of named variable are permitted: *immutable named variables*, which have "`?`" as a prefix followed by the variable name (e.g. `?item`) and *mutable named variables* that are prefixed with "`??`" (e.g. `??amount`).

Each named variable is in scope for the duration of a conversation and is associated with values as it is matched against the actual fields in messages that trigger transitions. Whenever a named variable is used in an immutable context,

```
<?xml version="1.0"?>

<protocol xmlns="http://acre.lill.is"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://acre.lill.is http://acre.lill.is/proto.xsd">

   <namespace>is.lill.acre</name>
   <name>acre-vickreyauction</name>
   <version>0.1</version>

   <states>
      <state name="start"/>
      <state name="awaiting_bid" />
      <state name="bid" />
      <state name="nobid"/>
      <state name="accepted"/>
      <state name="rejected"/>
   </states>

   <transitions>
      <transition performative="cfp"
                  from-state="start"
                  to-state="awaiting_bid"
                  sender="?initiator"
                  receiver="?bidder"
                  content="bidfor(?item)" />
      <transition performative="propose"
                  from-state="awaiting_bid"
                  to-state="bid"
                  sender="?bidder"
                  receiver="?initiator"
                  content="bid(?item,?amount)" />
      <transition performative="propose"
                  from-state="awaiting_bid"
                  to-state="nobid"
                  sender="?bidder"
                  receiver="?initiator"
                  content="nobid(?item)" />
      <transition performative="accept-proposal"
                  from-state="bid"
                  to-state="accepted"
                  sender="?initiator"
                  receiver="?bidder"
                  content="bid(?item,?amount)" />
      <transition performative="reject-proposal"
                  from-state="bid"
                  to-state="accepted"
                  sender="?initiator"
                  receiver="?bidder"
                  content="bid(?item,?amount)" />
   </transitions>
</protocol>
```

Fig. 5. ACRE XML Representation of the Vickrey Auction Protocol

it may match any content if has not already acquired a value. However, once it has been matched to a value, it may only match that value for the duration of the conversation. For example, Figure 5 uses `?initiator` to denote the agent that begins the Vickrey Auction with the sending of the initial call for proposals. Initially this variable does not have a value associated with it, so it may match the name of the relevant agent (e.g. "agent1"). However, once this has been done, the variable `?initiator` may only match the value "agent1" for the remainder of the conversation.

In some situations it is desirable to have variables whose values may change as the conversation progresses. For that reason, mutable named variables are also facilitated. The difference between a mutable and immutable named variable is that a mutable named variable can match against any content, regardless of whether it previously has a value associated with it. When this occurs, the existing value is overwritten with the new value that has been matched.

The motivation behind the use of mutable variables can be seen by examining the Vickrey Auction protocol shown above. This is a one-shot auction, so immutable variables are sufficient. However, implementing an iterated auction is made far more complex if all variables are immutable. The second transition shown in Figure 5 would be unsuitable for this, as the `?amount` variable is restricted to match only whatever the first value it is matched against. By changing the content field of this transition to `bid(?item,??amount)`, the variable relating to the amount is used in mutable context and so its value may change (although the item being bid for must remain the same). Thus each time this transition is triggered, the `amount` variable acquires a new value: namely the amount of the latest bid that was submitted. This can later be referred to using texttt?amount by the other agent that wishes to accept or reject the bid. A similar usage can be seen in the example presented in Section 6.

6 ACRE Conversation Example

This section presents an example of how an ACRE conversation may progress. The example is based on the "Process Documents" protocol shown in Figure 6. This protocol is designed for a system where text documents must undergo some form of processing. The Initiator of the protocol is capable of performing this processing, though it must be made aware of which documents to work on by the Respondent.

Initially, the Initiator informs the Respondent that it is ready to process documents, to which the Respondent replies with a request to process a particular document. The Initiator may either process the document and inform the Respondent of this, or refuse to carry out this processing. In the former case, the Respondent will send the next document for processing and continue to do this until the Initiator eventually refuses.

Here, the Agent UML description shown to the left of Figure 6 is converted to the FSM shown on the right. The dashed line surrounding the "Start" state indicate that this is the current state initially.

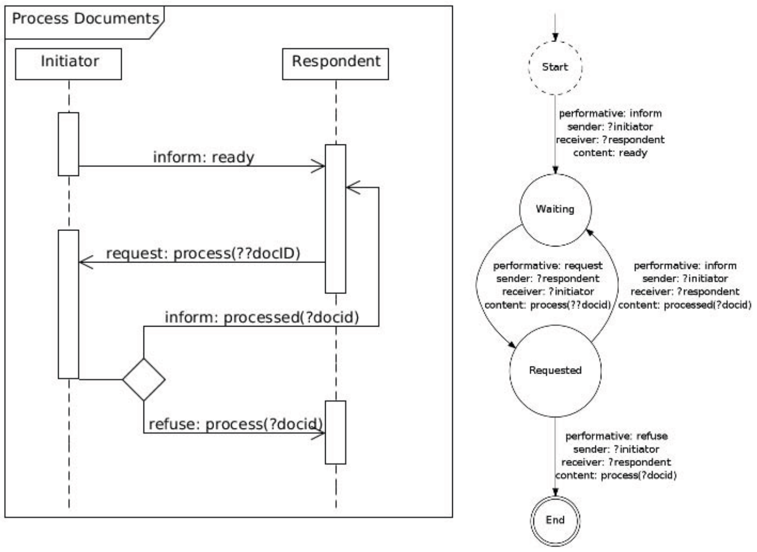


Fig. 6. Process Documents Protocol

The first transition contains constant values for both the performative and the content. This means that any message matching that transition must have those exact values in those fields. The message sender and receiver are variables, and since there are not yet any bindings associated with the conversation, these will match any agent identifiers in those fields in the message.

A message that will trigger this initial transition may look as follows (some unimportant fields have been omitted for clarity):

```
(inform
   :sender    processor
   :receiver  manager
   :content   ready
)
```

This results in the state of the conversation changing to "Waiting", as shown in Figure 7. Because the `?initiator` and `?respondent` matched against the constants "processor" and "manager" respectively in the message, these bindings are associated with the conversation. As these variables are used in immutable context throughout, they must match their exact bound values for the remainder of the conversation. This is indicated in Figure 7 by replacing the variables with these values. Angle brackets have been placed around each of the replacement values in order to emphasise this.

The transition from the "Waiting" state can now only be triggered by a message sent by the manager agent to the processor agent with the "request"

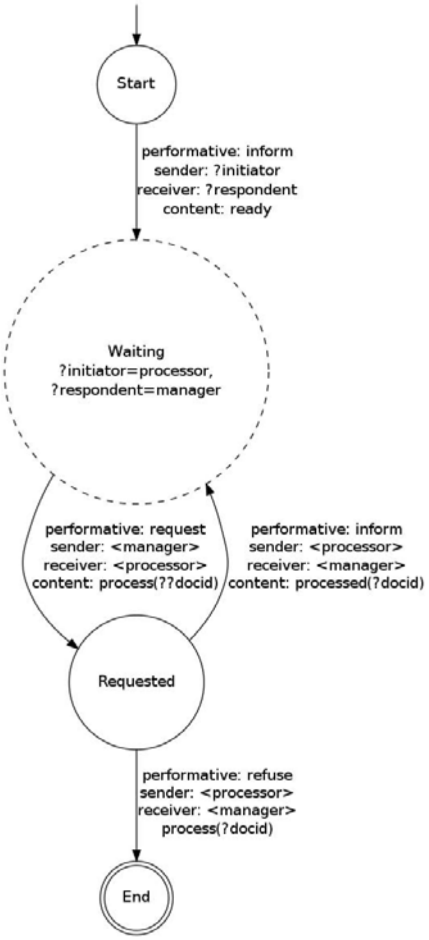


Fig. 7. Process Documents protocol in the "Waiting" state

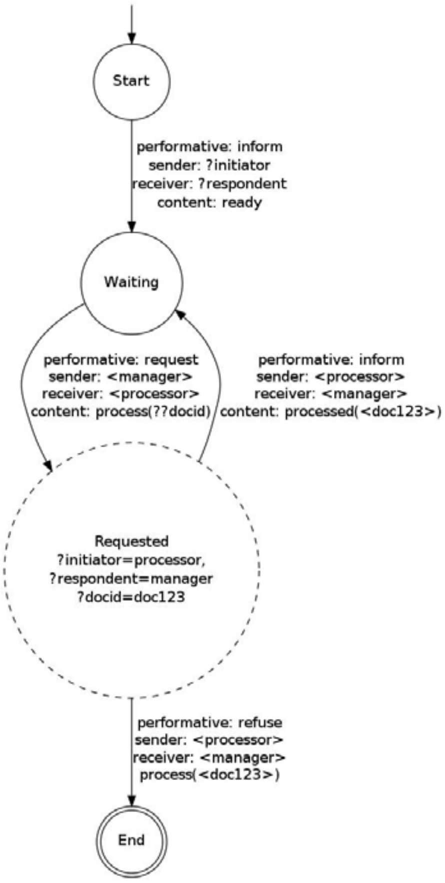


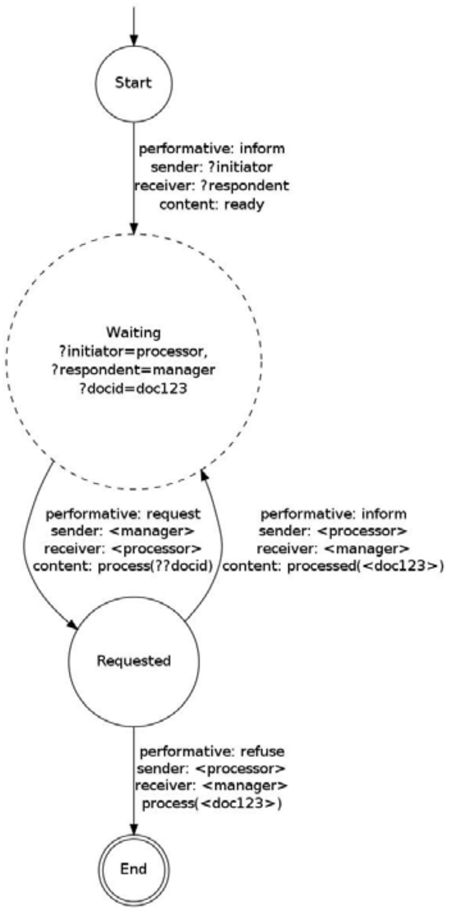
Fig. 8. Process Documents protocol in the "Requested" state

performative. The content must also match the transition, though with the use of the `??docid` variable, there is some flexibility in the values that can be matched.

In the next stage, the manager asks the processor to process the document with the identifier "doc123". This is done by means of the following message:

```
(request
   :sender    manager
   :receiver  processor
   :content   process(doc123)
)
```

As this message matches the transition, the conversation moves from the "Waiting" state to the "Requested" state, as shown in Figure 8.



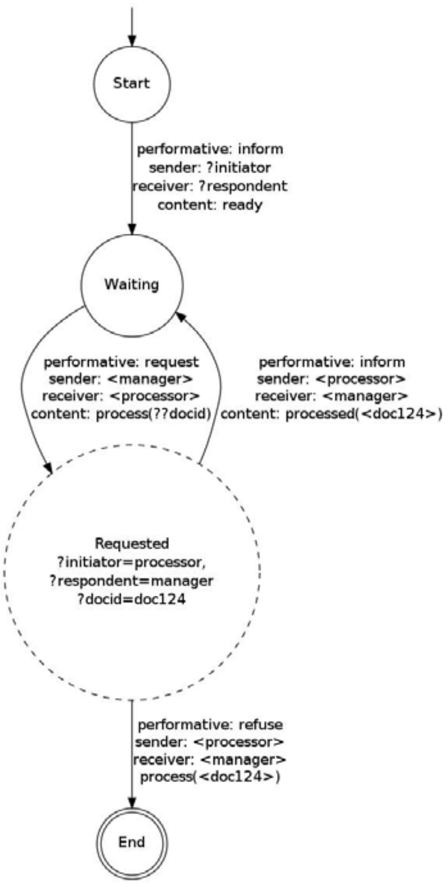


Fig. 9. Process Documents protocol in the "Waiting" state for the second time

Fig. 10. Process Documents protocol in the "Requested" state for the second time

At this point, the `?docid` variable has also acquired a binding, so this is replaced in all transitions using it in an immutable context. Again, this is indicated by the value being contained within angle brackets in Figure 8. At this point, the processor agent must either inform the manager that document "doc123" has been processed (thus returning the conversation to the "Waiting" state) or refusing to process that document. In each case, it is only the document identifier that has previously been bound to the `?docid` variable that may be used. Refusing to process a different document identifier would result in the conversation failing as the message could not match an available transition.

If the processor agrees to process the document and informs the manager of its completion, the conversation returns to the "Waiting" state, as shown in Figure 9. Unlike the first time the conversation was in this state, this time the `?docid` variable has got a binding associated with it. However, the transition between "Waiting" and "Requested" uses this variable in a mutable context, meaning that it can match against any value contained in the next message. Thus the manager can ask the processor to process any document it wishes. In contrast, as noted previously, when moving from the "Requested" state, the processor is restricted to only discussing the identifier of the last document it was asked to process.

If the manager requests that the processor processes document "doc124", the conversation returns to the "Requested" state, as shown in Figure 10. The `?docid` variable has now acquired an updated binding that is now applied to all the transitions using that variable in an immutable context. At this stage, the processor agent may repeat the cycle by processing the document and informing the manager of this, or it may end the conversation by refusing to perform the processing, thus entering the "End" state.

7 Language Integration in Agent Factory

Agent Factory is an extensible, modular and open framework for the development of multi agent systems [2]. It supports a number of agent programming languages, including AFAPL/ALPHA [18,19], AFAPL2 [20], AF-TeleoReactive (based on [21]) and AF-AgentSpeak (an implementation of AgentSpeak(L) [8]). The specific use of ACRE from within AFAPL2 agents has been discussed in a previous paper [16].

Agent Factory's implementation of these agent programming languages are based on its *Common Language Framework* (CLF) whereby the way in which sensors, actions and modules are implemented have been standardised across the various languages. This greatly facilitates the integration of additional services into each language, as the core components will be shared. This is the case for ACRE, where minimal effort is required to add support for further languages once one integration has been completed.

The ACRE Architecture consists of a number of components, some of which are platform-independent and some of which require some work to be ported to other platforms and agent programming languages.

7.1 Protocol Manager

The *Protocol Manager* (PM) is a platform-independent component that is tasked with making protocols available to agents. When an agent identifies the URL of an ACRE protocol definition it will send this to the PM, which will load the protocol, verify it against the appropriate schema and make it available for interested agents to use. It is also capable of accessing online ACRE repositories that may contain multiple protocol definitions in a centralised location. A repository definition file lists the protocols that it has available. Typically, one PM will exist on an agent platform, so any protocol located by any agent will be shared amongst all agents on the platform (within Agent Factory, the PM is a shared Platform Service). However, there is no technical barrier to individual agents having their own PMs if desired. Previously loaded protocols are also stored locally so that they can be recovered in the event of a platform failure or restart.

7.2 Conversation Manager

Whereas the PM is shared amongst agents, each agent has its own *Conversation Manager* (CM), which is used to keep track of the conversations the agent is involved in. The CM monitors both incoming and outgoing communication and matches each message to an appropriate conversation, following the algorithm outlined in Section 4. By monitoring the CM, an agent can gain data that can be used to reason about ongoing conversations and the messages it sends and receives. The CM is also platform-independent.

7.3 Agent/ACRE Interface

The *Agent/ACRE Interface* (AAI) is specific to the platform and agent programming language being used. This is designed to facilitate the interaction between the agents and the ACRE components mentioned above. The AAI has two distinct principal roles:

– To enable an agent to inform the PM and CM of information it holds.
– To provide the agent with information about the status and activity of the CM and PM.

In the former case, an agent must be capable of informing the PM of the location of any protocols that it wishes to use. This information may originally come from another agent with which it wishes to communicate. The CM requires access to the inbox and outbox of the agent also, so the AAI must provide this service also.

The key role of the AAI is making information about its own communication available to agents. Within Agent Factory, this is done in two complementary ways: *knowledge sensors* and *event sensors*.

A *knowledge sensor* is a sensor that runs on each interpreter cycle of the agent, and provides information on the current state of conversations and protocols. This information currently consists of:

– What protocols are already loaded (PM).
– For each conversation in which the agent is a participant:
– The protocol each conversation is following (CM)
– The identity of the other participating agent in each conversation (CM)
– The current state of each conversation (CM)
– The current status of each conversation (CM)

In addition to these, *event sensors* inform the agent whenever events are raised by the PM or the CM. Events currently handled include:

– A new protocol has been loaded (PM)
– A new conversation has begun (CM)
– A conversation has advanced (CM)
– A conversation has ended (CM)
– An error has occurred in a conversation (CM)

In addition to the basic role of information passing, an AAI may augment the capabilities of a language by leveraging the data available from the CM or the PM. For instance, the AAI built for the Agent Factory CLF provides an action of the form `advance(conversation-id,performative,content)` whereby an agent can advance a specific conversation while providing minimal information. The details about the other participating agent (including its address) are taken from the CM, along with the protocol identifier and content language. Further features in this vein are left for future work.

8 Conclusions and Future Work

This paper presents an outline of the ACRE conversation reasoning system, how it models protocols and conversations, and how it is integrated into the Agent Factory multi agent framework. Although currently only used with Agent Factory, it is intended that ACRE will be used in conjunction with several other agent programming frameworks and the languages they support. ACRE has been designed to be as language-independent and platform-independent as possible. Despite this, it will be necessary to adapt the system to frameworks and languages other than Agent Factory and the agent programming languages it supports.

Aside from the integration of ACRE into other platforms, future focus will be on the reasoning about conversations at the agent deliberative level and the information that ACRE will need to provide in order to facilitate this. One are of focus will be to allow the grouping of conversations into related groups. As ACRE protocols are limited to two participants, it is necessary to allow agents to relate conversations to each other. One such example will be in a situation where an agent is conducting an auction and has issued a call for proposals to multiple agents. At present, each of these initiates a separate conversation that must be managed by the agent using its own existing capabilities. However, grouping conversations at the Conversation Manager level would allow events to be raised to inform the agent that all conversations in the group had reached the state where a proposal had been received, or left the state where a proposal had been solicited. The key point is the provision of sufficient information for agents to use their deliberative reasoning capabilities in conjunction with the information emanating from the Conversation and Protocol Managers.

In addition, it is intended to explore the ways in which having access to an ACRE layer will add to the native messaging capabilities of other programming languages. This includes the ability to directly advance a conversation, as alluded to in Section 7, as well as specifically initiating conversations (rather than relying on message matching on the part of the Conversation Manager). The possibilities are likely to vary with different agent programming languages and it is intended to add these features as appropriate.

The availability of cross-platform communication tools such as ACRE can only aid interoperability between distinct agent platforms, toolkits and programming languages.

References

1. Labrou, Y.: Standardizing agent communication. Multi-Agents Systems and Applications (Advanced Course on Artificial Intelligence), 74–97 (2001)
2. Collier, R.W., O'Hare, G.M.P., Lowen, T., Rooney, C.: Beyond Prototyping in the Factory of Agents. In: Mařík, V., Müller, J.P., Pěchouček, M. (eds.) CEEMAS 2003. LNCS (LNAI), vol. 2691, p. 383. Springer, Heidelberg (2003)
3. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an Agent Communication Language. In: Proceedings of the Third International Conference on Information and Knowledge Management, Gaithersburg, MD, pp. 456–463 (1994)
4. Poslad, S., Buckle, P., Hadingham, R.: The FIPA-OS Agent Platform: Open Source for Open Standards. In: Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (PAAM 2000), Manchester, p. 368 (2000)
5. Foundation for Intelligent Physical Agents: FIPA Communicative Act Library Specification (2002)
6. Foundation for Intelligent Physical Agents: FIPA Interaction Protocol Library Specification (2000)
7. Bellifemine, F., Caire, G., Trucco, T., Rimass, G.: Jade Programmer's Guide (2007)
8. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming multi-agent systems in AgentSpeak using Jason. Wiley Interscience, New York (2007)
9. Collier, R.W., Ross, R., O'Hare, G.M.P.: A Role-Based Approach to Reuse in Agent-Oriented Programming. In: AAAI Fall Symposium on Roles, an Interdisciplinary Perspective (Roles 2005), Arlington, VA, USA (2005)
10. Barbuceanu, M., Fox, M.S.: COOL: A language for describing coordination in multi agent systems. In: Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-1995), pp. 17–24 (1995)
11. Cost, R.S., Finin, T., Labrou, Y., Luan, X., Peng, Y., Soboroff, I., Mayfield, J., Boughannam, A.: Jackal: a Java-based Tool for Agent Development. Working Papers of the AAAI 1998 Workshop on Software Tools for Developing Agents, AAAI Press, Menlo Park (1998)
12. Bradshaw, J.M., Dutfield, S., Benoit, P., Woolley, J.D.: KAoS: Toward an industrial-strength open agent architecture. Software Agents, 375–418 (1997)
13. Cost, R.S., Chen, Y., Finin, T., Labrou, Y., Peng, Y.: Modeling agent conversations with colored petri nets. In: Workshop on Specifying and Implementing Conversation Policies, Third International Conference on Autonomous Agents (Agents 1999), Seattle, pp. 59–66 (1999)
14. Parunak, H.: Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis. In: Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS) (1996)

15. Foundation for Intelligent Physical Agents: FIPA ACL Message Structure Specification (2002)
16. Lillis, D., Collier, R.W.: ACRE: Agent Communication Reasoning Engine. In: 3rd International Workshop on Languages, Methodologies and Development Tools for Multi Agent SystemS (LADS 2010), Lyon (2010)
17. Braubach, L., Pokahr, A.: Goal-Oriented Interaction Protocols. In: Petta, P., Müller, J.P., Klusch, M., Georgeff, M. (eds.) MATES 2007. LNCS (LNAI), vol. 4687, pp. 85–97. Springer, Heidelberg (2007)
18. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. Phd thesis, University College Dublin (2001)
19. Collier, R., Ross, R.J., O'Hare, G.M.P.: Realising Reusable Agent Behaviours with ALPHA. In: Eymann, T., Klügl, F., Lamersdorf, W., Klusch, M., Huhns, M.N. (eds.) MATES 2005. LNCS (LNAI), vol. 3550, pp. 210–215. Springer, Heidelberg (2005)
20. Muldoon, C., O'Hare, G.M.P., Collier, R.W., O'Grady, M.J.: Towards Pervasive Intelligence: Reflections on the Evolution of the Agent Factory Framework, vol. ch.6, pp. 187–212. Springer US, Boston (2009)
21. Nilsson, N.J.: Teleo-Reactive Programs for Agent Control. Journal of Artificial Intelligence Research 1, 139–158 (1994)

Exploiting Agent-Oriented Programming for Developing Future Internet Applications Based on the Web: The JaCa-Web Framework

Mattia Minotti, Alessandro Ricci, and Andrea Santi

DEIS, University of Bologna
Via Venezia 52, Cesena (FC), Italy
{a.ricci,a.santi}@unibo.it

Abstract. Besides being suitable for tackling Distributed Artificial Intelligence problems, we argue that agent-oriented programming languages and multi-agent programming technologies provide an effective level of abstraction for tackling the design and programming of complex software systems in general. Internet applications based on the Web are an important example of such systems. Following the cloud computing perspective, these kinds of applications will more and more replace desktop applications, exploiting the Web infrastructure as a common distributed operating system. We argue that the development of these applications raises challenges that cannot be effectively tackled by mainstream programming paradigms, such as the object-oriented one, and could be effectively faced – instead – by an agent-oriented approach. Accordingly, in this paper we describe JaCa-Web, a framework that allows for applying agent-oriented programming technologies – in particular JaCa, which integrates *Jason* agent programming language and CArtAgO environment technology – to the development of advanced Web client applications.

1 Introduction

The value of Agent-Oriented Programming (AOP) [23], including multi-agent systems technology, is often remarked and evaluated in the context of Artificial Intelligence (AI) and Distributed AI problems. This is evident, for instance, by considering existing agent programming languages (see [4,6] for comprehensive surveys) – whose features are typically demonstrated by considering AI toy problems such as block worlds and alike. Besides this view, we argue that the level of abstraction introduced by AOP is effective for organising and programming software applications in general, starting from those programs that involve aspects related to reactivity, asynchronous interactions, concurrency, up to those involving different degrees of autonomy and intelligence.

In that context, an important example is given by future Internet applications based on the Web. The Web has witnessed an extraordinary evolution in recent years, starting from being a distributed infrastructure for producing and accessing to hyper-text documents and becoming nowadays more and more the reference platform for running distributed Internet-based applications. *Rich Internet Applications* [7] are a result of such an evolution, being web-based applications which share more and more features with desktop applications, combining their better user experience with all the benefits provided by the Web, such as distribution, openness and accessibility. These kinds of applications are a fundamental brick for the *cloud computing* perspective [15], whereby shared resources, software and information are provided to computers and other devices on demand through the network, typically exploiting the Web – a simple example is given by the Google apps[1]. A main point in this evolution is the role of the client (browser), which is no more restricted to getting inputs from the user and visualising HTML pages only, but supporting the execution of part of the applications, which continuously interact with the server side and – more generally – services on the network. This change however brings also a substantial enhancement of the complexity in the development of the Web applications, and it is clear that basic enabling mechanisms (such as AJAX technology or JavaScript scripting language) are not enough when we deal with applications in the large.

We argue that agent-orientation provides a suitable level of abstraction to design and develop these kinds of applications. In this paper we show this idea in practice by describing an agent-oriented framework called JaCa-Web that allows for programming and executing Internet applications based on the Web, focussing in particular on the client side. JaCa-Web is based on the JaCa platform, which integrates the *Jason* agent programming language and CArtAgO environment programming framework. Besides describing the platform, our aim here is to discuss the key points that make JaCa and – more generally – agent-oriented programming a suitable paradigm for tackling main complexities of software applications, advanced Web applications in this case, that – we argue – are not properly addressed by mainstream programming languages, such as object-oriented ones. In that, this work can be considered an extension of a previous one [13], where a Java-based agent framework called simpA [22] is used.

The remainder of the paper is organised as follows. First, we provide a brief overview of JaCa (Section 2) programming model and platform. Then, we discuss the use of JaCa for developing Web client applications (Section 3), remarking the advantages compared to existing state-of-the art approaches. To evaluate the approach we describe the design and implementation of a case study (Section 4). Finally, we present related and future works (Section 5) and concluding remarks (Section 6).

2 Agent-Oriented Programming with JaCa

An application in JaCa is designed and programmed as a set of agents which work and cooperate inside a common environment. Following the A&A conceptual model [14,21], the environment is organised as a set of *workspaces* – possibly distributed over the network – populated by a dynamic set of *artifacts*, that are

[1] `http://www.google.com/apps/`

the environment basic building blocks representing resources and tools shared and used by agent to do their work. Programming the application means then programming the agents on the one side, as goal-oriented entities encapsulating the logic of control of the tasks that must be executed, and the artifacts on the other side, as first-class environmental abstraction providing the actions and functionalities exploited by the agents to do their tasks. It is worth remarking that, differently from traditional agent-oriented programming technologies, the environment here plays a key role as a *programmable* part of the system [20], useful to encapsulate those non-autonomous services that can be used by agents at runtime – which include also the interaction with the external environment. In this approach, agents can interact and communicate both directly, through speech acts and ACL, and indirectly, through the use of the shared environment.

2.1 Programming the Agents

In JaCa, *Jason* [5] is adopted as the programming language to implement and execute the agents. Being a concrete implementation of an extended version of AgentSpeak(L) [18], *Jason* adopts a BDI (Belief-Desire-Intention)-based computational model and architecture to define the structure and behaviour of individual agents. In that, agents are implemented as reactive planning systems: they run continuously, reacting to events (e.g., perceived changes in the environment) by executing plans given by the programmer. Plans are courses of actions that agents commit to execute so as to achieve their goals. The pro-active behaviour of agents is possible through the notion of goals (desired states of the world) that are also part of the language in which plans are written. Besides interacting with the environment, *Jason* agents can communicate by means of speech acts.

2.2 Programming the Environment

On the environment side, CArtAgO [20] is adopted as a framework to program and run the artifact-based environments. Artifacts are, on the one side, a first-class programming abstraction for the environment developer to modularise the environment functionalities, by programming the different kinds of artifacts that can be instantiated and used at runtime by the agents; on the other side, they are first-order entities from the agent view point, i.e. the basic bricks that populate the agent world and that agents typically exploit as resources and tools – possibly shared – to do their job.

In order to be used by the agents, each artifact provides of a usage interface composed by a set of *operations* and *observable properties*. Operations correspond to the actions that the artifact makes it available to agents to interact with such a piece of the environment. Operations are executed by the artifact *transactionally*, and only one operation can be in execution at a time, like in the case of monitors in concurrent programming. Complex operations, however, can be structured in multiple transactional steps that can be interleaved, so as to allow for the concurrent execution of operations. Observable properties define the observable state of the artifact, which is represented by a set of information items whose value (and value change) can be perceived by agents as percepts. Besides observable properties, the execution of operations can generate signals perceivable by agents as percepts, too. Finally, as a principle of composability, artifacts can be assembled together by a link mechanism, which allows for an artifact to execute operations over another artifact.

The notion of workspace – as a logical cointainer of agents and artifacts – is used to define the topology of complex environments, that can be organised as multiple sub-environments (workspaces), possibly distributed over the network. By default, each workspace contains a predefined set of artifacts created at boot time, providing basic functionalities: examples are the workspace artifact, which provides operations (actions) to create, lookup, link together artifacts, the node artifact, providing operations to join multiple workspaces, the console artifact, to print message on the console. An agent can work simultaneously in multiple workspaces, which may reside both in the same network node and in remote (distributed) nodes.

CArtAgO provides a Java-based API to program the types of artifacts that can be instantiated and used by agents at runtime. So an *object-oriented* data-model is adopted to define the data structures in actions, observable properties and events.

2.3 Multi-agent System View: Putting Agents and Artifacts Together

JaCa integrates *Jason* and CArtAgO so as to make it seamless the use of artifact-based environments by *Jason* agents.

To this purpose, first, the overall set of external actions that a *Jason* agent can perform is determined by the overall set of artifacts that are actually available in the workspaces where the agent is working. So, the action repertoire is dynamic and can be changed by agents themselves by creating, disposing artifacts.

Then, the overall set of percepts that a *Jason* agent can observe is given by the observable properties and observable events of the artifacts available in the workspace at runtime. Actually an agent can explicitly select which artifacts to observe, by means of a specific action called *focus*. By observing an artifact, artifacts' observable properties are directly mapped into beliefs in the belief-base, updated automatically each time the observable property changes its value. So a *Jason* agent can specify plans reacting to changes to beliefs that concern observable properties or can select plans according to the value of beliefs which refer to observable properties. Artifacts' signals instead are not mapped into the belief base, but processed as non persistent percepts possibly triggering plans—like in the case of message receipt events.

Finally, the *Jason* data-model – essentially based on Prolog terms – is extended to manage also (Java) objects, so as to work with data exchanged by performing actions and processing percepts.

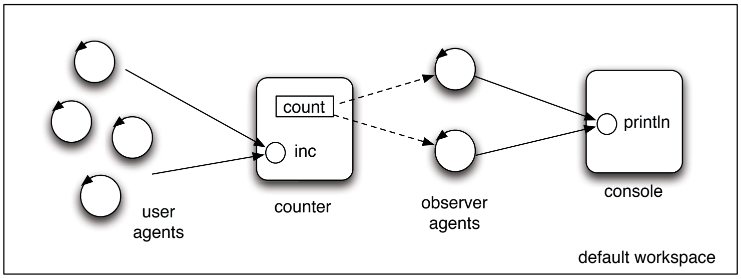


Fig. 1. A simple JaCa program with some user and observer agents which interact by using and observing a shared `Counter` artifact

```
// MAS main
MAS example {
  environment:
  c4jason.CartagoEnvironment

  agents:
  user user agentArchClass
           c4jason.CAgentArch #4;
  observer observer agentArchClass
           c4jason.CAgentArch #2;
}
```

```
// Counter artifact template
public class Counter extends Artifact {
  void init(){
    defineObsProperty("count",0);
  }
  @OPERATION void inc(){
    ObsProperty prop = getObsProperty("count");
    prop.updateValue(prop.intValue()+1);
    signal("tick");
  }
}
```

```
// user agent code
!create_and_use.

+!create_and_use : true
  <- !setupTool(Id);
     inc;
     inc [artifact_id(Id)].
+!setupTool(C): true
  <- makeArtifact("c0",
     "Counter",[],C).
-!setupTool(C): true
  <- lookupArtifact("c0",C).
```

```
// observer agent code
!observe.

+!observe : true
  <- ?myTool(C);  // discover the tool
     focus(C).
+count(V)
  <- println("observed new value: ",V).
+tick [artifact_name(Id,"c0")]
  <- println("perceived a tick").
+?myTool(CounterId): true
  <- lookupArtifact("c0",CounterId).
-?myTool(CounterId): true
  <- .wait(10); ?myTool(CounterId).
```

Fig. 2. The complete source code of the JaCa program, including the *Jason* MAS main file, the source code of the `Counter` artifact template and the source code of the user and observer agents

Just to have a taste of the approach, a toy example of JaCa program is depicted in Fig. 1, in which four agents `users` interact with two agents `observer` by means of a shared `counter` artifact, instance of the `Counter` artifact template. Each user agent tries to create the `counter` artifact by means of the `makeArtifact` action – which is an operation provided by the `workspace` artifact, available in every workspace – and then to use it by executing twice the `inc` action – which corresponds to the `inc` artifact operation. Actually only one user agent would succeed in creating the artifact: the other user agents simply retrieve the artifact

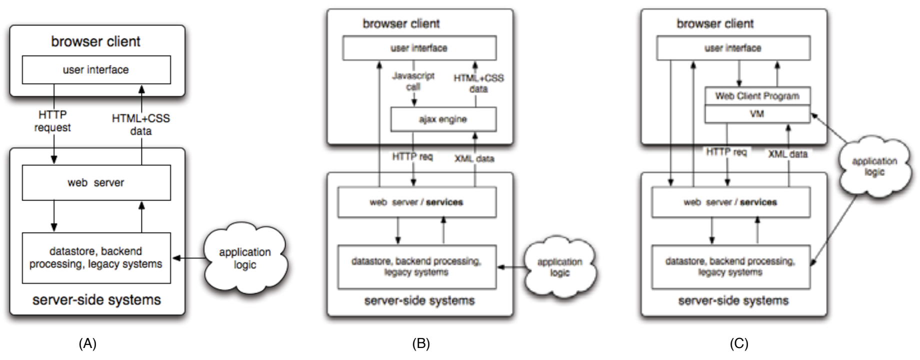


Fig. 3. Evolution of the architecture of Web Applications

identifier in the repair plan `-!setupTool <- ...`, which is executed as soon as the `+!setupTool <- ...` fails[2]. Each observer agent first discovers the `counter` artifact, then starts observing it: as soon as the `count` observable property of the artifact is changed or a `tick` observable event is generated, observers react and print on the console artifact a message.

A full description of *Jason* language/platform and CArtAgO framework – and their integration – is out of the scope of this paper: the interested reader can find details in literature [20,19] and on *Jason* and CArtAgO open-source web sites[3,4].

3 Applying JaCa to the Web: The JaCa-Web Framework

In this section, we describe how the features of JaCa can be exploited to program complex Web client applications, providing benefits over existing approaches. First, we sketch the main complexities related to the design and programming of next-generation Web applications; then, we describe how these are addressed by JaCa-Web, which is a framework on top of JaCa to develop such a kind of applications.

3.1 Developing Future Internet Web-Based Applications: Challenges

As mentioned in the introduction, since its conception the role of the Web has substantially evolved, starting from being a distributed infrastructure for producing and accessing to hyper-text documents and becoming nowadays the standard de facto platform for running distributed Internet-based applications. Even as a platform for developing and deploying applications, it is possible to devise in the

[2] For these user agents, since it is not possible to create multiple artifacts with the same logic name, the `makeArtifact` action fails.

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

[4] `http://cartago.sourceforge.net`

state-of-the-art an evolution of Web application architectures, as depicted in Fig. 3. In the first architectural model – still used in Web 1.0 applications – all the application logic is on the server side: the client side – represented by the browser – is responsible of visualising HTML pages and get inputs from the user. By referring to the MVC (Model View Controller) architecture, both the Model and the Controller parts are on the server side, leaving only the View part on the client side. Every user input action causes the execution of a new HTTP synchronous request, processed on the server side by the controller (such as a Servlet) accessing to the Model (a data-base or some components in a component container) and generating a new HTML page to be sent back to the client.

A first extension to this model (point B in the figure) is given by the introduction of proper set of technologies – AJAX is a main example – that allow for executing *asynchronous* calls from the client to the server, so as to realise applications with a more reactive user interface, more similar to the one adopted by desktop applications. In particular, such Web applications allow the client to send multiple concurrent requests asynchronously, avoiding complete page reload and keeping the user interface live and responding. Periodic activities within the client-side of the applications can be performed in the same fashion, with clear advantages in terms of perceived performance, efficiency and interactivity. So the application logic is still on the server side, however the model of interaction radically changes, so that both on the server side and the client side applications need to manage asynchronous events and concurrent computations.

The next final step (point C) extends and generalises the previous one by distributing the application logic between the client and the server, bringing then part of the Controller and the Model on the client side and exploiting the synchronous plus asynchronous model of interaction to glue the parts together, besides the goal of a better UI. So the more complex Web apps are considered, the more the application logic put on the client side becomes richer, eventually including asynchronous interactions – with the user, with remote services – and possibly also concurrency – due to the concurrent interaction with multiple remote services. This step requires the availability on the client side – inside the browser – of proper technologies that would allow for the execution of full-fledged applications, which need to flexibly interact with both the user locally and with the Web server and services remotely.

The direction of decentralising responsibilities to the client, and eventually improving the capability of working offline, is evident also by considering the new HTML standard 5.0, which enriches the set of API and features that can be used by the Web application on the client side[5]. Among the others, some can have a strong impact on the way an application is designed: it is the case of the Web Worker mechanism[6], which makes it possible to spawn background workers running scripts in parallel to their main page, allowing for thread-like operation with message-passing as the coordination mechanism. Another one is

[5] `http://dev.w3.org/html5/spec/`

[6] `http://www.whatwg.org/specs/web-workers/current-work/`

cross-document messaging[7], which defines mechanisms for communicating between browsing contexts in HTML documents.

Besides devising enabling mechanisms, a main issue is then how to design and program of these kinds of applications. A basic and standard way to realise the client side of Web app is to embed in the page scripts written in some scripting language – such as JavaScript. Originally such scripts were meant just to perform check on the inputs and to create visual effects. The problem is that scripting languages – like JavaScript – have not been designed for programming in the large, so using them to organise, design, implement complex programs is hard and error-prone. To address the problems related to scripting languages, higher-level approaches have been proposed, based on frameworks that exploit mainstream object-oriented programming languages. A main example is Google Web Toolkit (GWT)[8], which allows for developing client-side apps with Java. This choice makes it possible to reuse and exploit all the strength of mainstream programming-in-the-large languages that are typically not provided by scripting languages—an example is strong typing. However it does not provide significant improvement for those aspects that are still an issue for OO programming languages, such as concurrency, asynchronous events and interactions, and so on. The same comment applies to solutions such as Microsoft Silverlight technology[9], which allows for developing Web client apps on top of the .NET Object-Oriented framework.

We argue then that these aspects can be effectively captured by adopting an agent-oriented level of abstraction and programmed by exploiting agent-oriented technologies such as JaCa: in next section we discuss this point in detail.

3.2 An Agent-Oriented Programming Approach Based on JaCa

By exploiting JaCa, we directly program the Web client application as a normal JaCa agent program, composed by a workspace with one or multiple agents working within an artifact-based environment including a set of pre-defined type of artifacts specifically designed for the Web context domain (see Fig. 4). Generally speaking, agents are used to encapsulate the logic of control and execution of the tasks that characterise the Web client app, while artifacts are used to implement the environment needed for executing the tasks, including those coordination artifacts that can ease the coordination of the agents' work. As soon as the page is downloaded by the browser, the application is launched – creating the workspace, the initial set of agents and artifacts.

Among the pre-defined types of artifact available in the workspace, two main ones are the *Page* artifact and the *HTTPService* artifact. *Page* provides a twofold functionality to agents: (i) to access and change the Web page, internally exploiting specific low-level technology to work with the DOM (Document Object Model) object, allowing for dynamically updating its content, structure, and

[7] `http://dev.w3.org/html5/postmsg/`
[8] `http://code.google.com/webtoolkit/`
[9] `http://www.silverlight.net/`

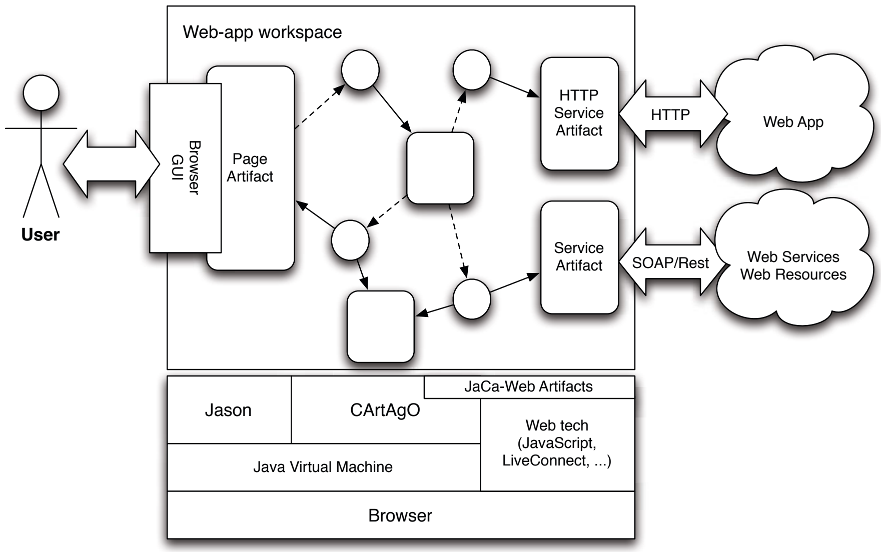


Fig. 4. An abstract overview of a JaCa-Web application

visualisation style; (ii) to make events related to user's actions on the page observable to agents as percepts. An application may either exploit directly *Page* or define its own extension with specific operations and observable properties linked to the specific content of the page. *HTTPService* provides basic functionalities to interact with a remote HTTP service, exploiting and hiding the use of sockets and low-level mechanisms. Analogously to *Page*, this kind of artifact can be used as it is – providing actions to do HTTP requests – or can be extended providing a higher-level application specific usage interface hiding the HTTP level.

So the artifact-based environment plays an important role here in situating the agents in the Web environment, so providing all the means to change the DOM, to interact with HTTP remote server and services, to perceive events produced by the user acting on the HTML page, and so on, but at a proper level of abstraction (actions and perceptions), hiding the low level mechanisms that concern the integration with the legacy technology.

To exemplify the description of these elements and of JaCa-Web programming in the overall, in the following we consider a toy example of Web client app, in which two agents are used to search for prime numbers up to a maximum value which can specified and dynamically changed by the user through the Web page. As soon as an agent finds a new prime number, a field on the the Web page reporting the total number of values is updated. Fig. 5 shows an overview of the main components of the program: the environment includes a Page artifact called `myPage`, an artifact called `numGen`, functioning as a number generator, shared and used by agents to get the numbers to verify, and two artifacts, `primeService1` and `primeService2`, providing the (same) functionality that is verifying if a

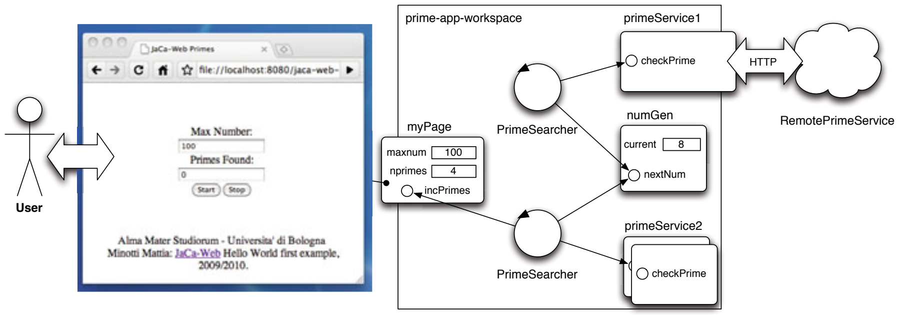


Fig. 5. An abstract overview of the JaCa-Web toy example described in the paper

number is prime. `myPage` is an instance of `MyPage` extending the basic *Page* artifact so as to be application specific, by: *(i)* defining an observable property `maxnum` whose value is directly linked to the related input field on the web page; *(ii)* generating `start` and `stop` signals as soon as the page button controls start and stop are pressed; *(ii)* defining an operation `incPrimes` that updates the output field of the page reporting the actual number of prime numbers found. `numGen` is an instance of the `NumGen` artifact (see Fig. 5), which provides an action `getNextNum` to generate a new number – retrieved as output (i.e. action feedback) parameter. The two prime number service artifacts provide the same usage interface, composed by a `checkPrime(num: integer)` action, which generates an observable event `is_prime(num: integer)` if the number is found to be prime. One artifact does the computation locally (`LocalPrimeService`); the other one, instead – which is an instance of `RemotePrimeService`, extending the pre-defined *HTTPService* artifact – provides the functionality by interacting with a remote HTTP service.

Fig. 6 shows the source code of one of the two prime searcher agents (on the right). After having set up the tools needed to work, the agent waits to perceive a `start` event generated by the page. Then, it starts working, repeatedly getting a new number to check – by executing a `getNextNum` – until the maximum number is achieved. The agent knows such a maximum value by means of the `maxnum` page observable property—which is mapped onto the related agent belief. The agent checks the number by performing the action `checkPrime` and then reacting to `is_prime(Num: integer)` event, updating the page by performing `incPrimes`. If a `stop` event is perceived – which means that the user pressed the stop button on the Web page – the agent promptly reacts and stops working, dropping its main intention.

3.3 Key Points

We have identified three key points that, in our opinion, represent main benefits of adopting agent-oriented programming and, in particular, the JaCa-Web programming model, for developing these kinds of applications.

```
public class MyPage extends PageArtifact {

  protected void setup() {
    defineObsProperty("maxnum",getMaxValue());
    linkEventToOp("start","click","startClicked");
    linkEventToOp("stop","click","stopClicked");
    linkEventToOp("maxnum","change","maxnumChange");
  }
  @OPERATION void incPrimes(){
    Elem el = getElementById("primes_found");
    el.setValue(el.intValue()+1);
  }
  @INTERNAL_OPERATION   private void startClicked(){
    signal("start");
  }
  @INTERNAL_OPERATION   private void stopClicked(){
    signal("stop");
  }
  @INTERNAL_OPERATION   private void maxnumChange(){
    updateObsProperty("maxnum",getMaxValue());
  }
  private int getMaxValue(){
    return getElementById("maxnum").intValue();
  }
}

public class RemotePrimeService extends HTTPService {

  @OPERATION void checkPrime(double n){
    HTTPResponse res =
           doHTTPRequest(serverAddr,"isPrime",n);
    if (res.getElem("is_prime").equals("true")){
      signal("is_prime",n);
    }
  }
}

public class NumGen extends Artifact {

  void init(){
    defineObsProperty("current",0);
  }
  @OPERATION
  void nextNum(OpFeedbackParam<Integer> res){
    int v = getObsProperty("current").intValue();
    updateObsProperty("current",++v);
    res.set(v);
  }
}
```

```
// Prime searcher agent

!setup.

+!setup
 <- lookupArtifact("MyPage",Page);
    focus(Page);
    makeArtifact("primeService1",
               "RemotePrimeService");
    makeArtifact("numGen","NumGen").

+start
 <- lookupArtifact("primeService1",S);
    focus(S);
    lookupArtifact("numGen",G);
    focus(G);
    !!checkPrimes.

+!checkPrimes
 <- nextNum(Num);
    !checkNum(Num).

+!checkNum(Num) :
   maxnum(Max) & Num <= Max
  <- checkPrime(Num);
       !checkPrimes.

+!checkNum(Num) :
  maxnum(Max) & Num > Max.

+is_prime(Num)
 <- incPrimes.

+stop
 <- .drop_intention(checkPrimes).
```

Fig. 6. *(Left)* Artifacts' definition in CArtAgO: `MyPage` and `RemotePrimeService` extending respectively `PageArtifact` and `HTTPService` artifact types which are available by default in JaCa-Web workspaces, and `NumGen` to coordinate number generation and sharing. *(Right)* *Jason* source code of a prime searcher agent.

First, agents are first-class abstractions for mapping possibly concurrent tasks identified at the design level, so reducing the gap from design to implementation. The approach allows for choosing the more appropriate concurrent architecture, allocating more tasks to the same kind of agent or defining multiple kind of agents working concurrently. This allows for easily programming Web client concurrent applications, that are able to exploit parallel hardware on the client side (such as multi-core architectures). In the example, two agents are used to fairly divide the overall job and work concurrently, exploiting the number generator artifact as a coordination tool to share the sequence of numbers. Changing the solution by using a single agent or more than two agents would not require any substantial change in the code.

Indeed, this simple example could be easily developed also using Java with flat *threads*. Threads, however, are a low-level OS mechanism notoriously not effective for hiding concurrency complexity when developing complex concurrent programs. The adoption of an agent level of abstraction makes it possible to reason, design and program using higher level concepts such as tasks and plans, which are provided as first-class programming constructs. The importance of having a proper level of abstraction is clearly manifested also by recent works about extending the Java concurrency library with proper general-purpose frameworks to hide as much as possible the use of threads. An example is given by the *executor* framework, available in the `java.util.concurrent` library since the 1.5 version of the Java Development Kit (JDK), where a logical notion of task is introduced.

Compared to existing OO concurrency frameworks, the computational model adopted by the *Jason* agent programming language – which is BDI based – provides a far richer support to task management, and – in particular – provides a clear model about how to integrate task-oriented behaviours and reactive event-driven behaviours—which is a well-known problem in concurrent programming [9]. This is the second key point. Agent architectures based on BDI-like reasoning cycle make it possible to straightforwardly integrate the management of asynchronous events generated by the environment – such as the input of the user or the responses retrieved from remote services – and the management of workflows of possibly articulated activities, which can be organised and structured in plans and sub-plans. This makes it possible to avoid the typical problems concerning the inversion of control [8] – caused by the use of callbacks to manage events – within multi-threaded programs. These problems are often referred ad *asynchronous spaghetti code*, that accounts for a lack of modularity in callback-based source code where the overall business logic concerning the execution of some task is fragmented into multiple call-back handlers that maybe executed asynchronously and possibly concurrently. In that case the programmer must *(i)* keep track of the overall task state by hand, using proper shared variables, and *(ii)* using low level mechanisms – such as locks – to avoid interferences in the case of concurrent handlers. In our case, these aspects can be designed and programmed at a higher-level of abstraction, in terms of reactive and pro-active structured plans.

In the prime searcher agent shown in the example, for instance, on the one hand we use a plan handling the `checkPrimes` goal to pro-actively search for prime numbers. The plan is structured then into a subgoal `checkNum` to process the number retrieved by interacting with the number generator. Then, the plan executed to handle this subgoal depends on the dynamic condition of the system: if the number to process is greater than the current value of the `maxnum` page observable property (i.e. of its related agent belief), then no checks are done and the goal is achieved; otherwise, the number is checked by exploiting a prime service available in the environment and the a new `checkPrimes` goal is issued to go on exploring the rest of the numbers. The user can dynamically change the value of the maximum number to explore, and this is promptly perceived by the agents which can change then their course of actions accordingly. On the other hand, reactive plans are used to process asynchronous events from the environment, in particular to process incoming results from prime services (plan `+is_prime(Num) <- ...`) and user input to stop the research (plan `+stop <- ...`).

Finally, the third key point concerns the strong separation of concerns which is obtained by exploiting the environment as first class abstraction. *Jason* agents, on the one side, encapsulate solely the logic and control of tasks execution; on the other side, basic low-level mechanisms to interact and exploit the Web infrastructure are wrapped inside artifacts, whose functionalities are seamlessly exploited by agents in terms of actions (operations) and percepts (observable properties and events). So the environment – represented by artifacts, in this case – is an effective mean to reuse and integrate object-oriented technologies, keeping the agent level of abstraction. Also, application specific artifacts – such as `NumGen` – can be designed to both encapsulate shared data structures useful for agents' work and regulate their access by agents, functioning as a coordination mechanism.

3.4 JaCa-Web: Implementation Details

In order to run JaCa-Web framework on existing Web clients, we exploited Java Applet technology, which provides a full-fledged Java Virtual Machine to execute the JaCa runtime. We could not use JavaScript, due to its single-threaded nature. Applets allow to transfer code from a server to a browser and have it executed within a controlled secure environment known as sandbox; in particular, *signed applets* drop much of the security constraints of the sandbox, for instance allowing Java classes to open their own connections towards multiple servers. Furthermore, the Java Virtual Machine invoked by the browser does not force any restriction on the number of threads that a program may spawn, thus providing a truly concurrent environment where to execute our application.

Since our model defines some client-side persistent entities and requires *Jason* + CArtAgO platforms, we also adopted *Java download extensions*[10] architecture for a one-time deployment of these parts of the Web application. This mechanism provides a simple, standard and scalable way to make custom APIs available to all applications running on the Java platform, therefore also applets executed on browser JVM. To exploit it, Web application dependencies, such as *Jason* and CArtAgO, have to be written in the manifest file of the main applets jar, along with references to URLs where these dependencies can be solved.

Concerning the access to the page DOM representation in the browser – which is provided by *Page* artifacts – we exploited the LiveConnect,[11] library that can

[10] `http://java.sun.com/docs/books/tutorial/ext/basics/download.html`
[11] `https://jdk6.dev.java.net/plugin2/liveconnect/`

be both used from the JVM multi-threaded execution context to invoke Javasctipt methods and, vice versa, from scripts to Java objects inside applet context. Liveconnect is included in Java Plug-in and it has been recently rewritten to provide an interoperability layer with improvements in terms of reliability, performance and cross-browser portability.

Finally, the *HTTP Channel* artifact does not exploit the HTTP protocol support provided by the browser, but relies instead on the functionalities offered by the Java standard library.

4 A Case Study

To stress the features of agent-oriented programming and test-drive the capabilities of the JaCa-Web framework, we developed a real-world Web application – with features that go beyond the ones that are typically found in current Web client app. The application is about searching products and comparing prices from multiple services, a "classic" problem on the Web.

We imagine the existence of N services that offer product lists with features and prices, codified in some standard machine-readable format. The client-side in the Web application needs to search all services for a product that satisfies a set of user-defined parameters and has a price inferior to a certain user-defined threshold. The client also needs to periodically monitor services so as to search for new offerings of the same product. A new offering satisfying the constraints should be visualised only when its price is more convenient than the currently best price. The client may finish its search and monitoring activities when some user-defined conditions are met—a certain amount of time is elapsed, a product with a price less than a specified threshold is find, or the user interrupts the search with a click on a proper button in the page displayed by the browser. Finally, if such an interruption took place, by pressing another button it must be possible to let the search continue from the point where it was blocked.

The characteristics of concurrency and periodicity of the activities that the client-side needs to perform make this case study a significant prototype of the typical Web client application. Typically these applications are realised by implementing all the features on the server side, without – however – any support for long-term searching and monitoring capabilities. In the following, we describe a solution based on JaCa-Web, in which responsibilities related to the long-term search and comparison are decentralised into the client side of the application, improving then the scalability of the solution – compared to the server-side solution – and the user experience, providing a reactive user interface and a desktop-like look-and-feel.

4.1 Application Design

The solution includes two kinds of agents (see Fig. 7): a *UserAssistant* agent – which is responsible of setting up the application environment and manage interaction with the user – and multiple *ProductFinder* agents, which are responsible

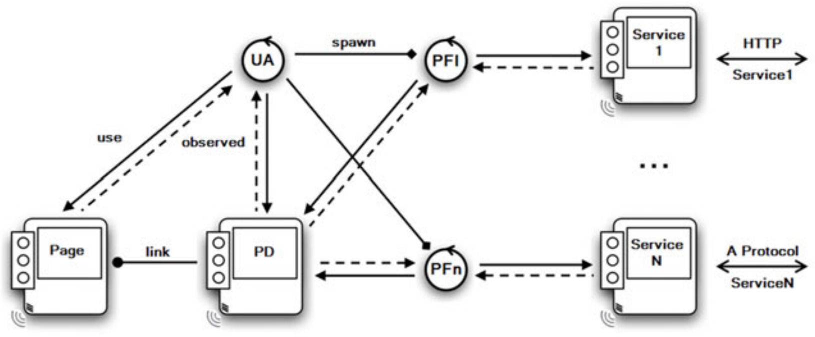


Fig. 7. The architecture of the client-side Web application sample in terms of agent, artifacts, and their interactions. `UA` is the *UserAgent*, `PF`s are the *ProductFinder* agents, `PD` is the *ProductDirectory* artifact and finally `Service`s are the *ProductService* artifacts

to periodically interact with remote product services to find the products satisfying the user-defined parameters. To aggregate data retrieved from services and coordinate the activities of the *UserAssistant* and *ProductFinder* we introduce a *ProductDirectory* artifact, while a *MyPage* page artifact and multiple instances of *ProductService* artifacts are used respectively by the *UserAssistant* and *ProductFinder* to interact with the user and with remote product services.

More in detail, the *UserAssistant* agent is the first agent booted on the client side, and it setups the application environment by creating the *ProductDirectory* artifact and spawning a number of *ProductFinder* agents, one for each service to monitor. Then, by observing the *MyPage* artifact, the agent monitors user's actions and inputs. In particular, the Web page provides controls to start, stop the searching process and to specify and change dynamically the keywords related to the product to search, along with the conditions to possibly terminate the process. Page events are mapped onto `start` and `stop` observable events generated by *MyPage*, while specific observable properties – `keywords` and termination conditions – are used to make it observable the input information specified by the user.

The *UserAssistant* reacts to these observable events and to changes to observable properties, and interacts with *ProductFinder* agents to coordinate the searching process. The interaction is mediated by the *ProductDirectory* artifact, which is used and observed by both the *UserAssistant* and *ProductFinder*s. In particular, this artifact provides a usage interface with operations to: *(i)* dynamically update the state and modality of the searching process – in particular `startSearch` and `stopSearch` to change the value of a `searchState` observable property – useful to coordinate agents' work – and `changeBasePrice`, `changeKeywords` to change the value of the base price and the keywords describing the product, which are stored in a `keyword` observable property; *(ii)* aggregate product information found by *ProductFinder*s – in particular `addProducts`, `removeProducts`, `clearAllProducts` to respectively add and remove a product, and remove all products found so far.

Besides searchState and keywords, the artifact has further observable properties, bestProduct, to store and make it observable the best product found so far.

Finally, each *ProductFinder*s periodically interact with a remote product service by means of a private *ProductService* artifact, which extends a *HTTPService* artifact providing an operation (requestProducts) to directly perform high-level product-oriented requests, hiding the HTTP level.

4.2 Implementation

The source code of the application can be consulted on the JaCa-Web web site[12], where the interested reader can find also the address of a running instance that can be used for tests. Here we just report a snippet of the *ProductFinder* agents' source code (Fig. 8), with in evidence *(i)* the plans used by the agent to react to changes to the search state property perceived from the *ProductDirectory* artifact - adding and removing a new search goal, and *(ii)* the plan used to achieve that goal, first getting the product list by means of the requestProducts operation and then updating the *ProductDirectory* accordingly by adding new products and removing products no more available. It is worth noting the use of the keywords belief – related to the keywords observable property of the *ProductDirectory* artifact – in the context condition of the plan to automatically retrieve and exploit updated information about the product to search.

```
// ProductFinder agent

...

+searchState("start")
  <- lookupArtifact("service1",Service); focus(Service); !!search.

+!search: keywords(Keywords)
  <- requestProducts(Keywords,ProductList);
     !processProducts(ProductList, ProductsToAdd, ProductsToRemove);
     addProducts(ProductsToAdd);
     removeProducts(ProductsToRemove);
     .wait({+keywords(_)},5000,_);
     !search.

+searchState("stop")
  <- .drop_intention(search).
```

Fig. 8. A snippet of *ProductFinder* agent's plans.

5 Related and Future Work

Several frameworks and bridges have been developed to exploit agent technologies for the development of Web applications. Main examples are the Jadex Webbridge [16], JACK WebBot [1] and the JADE Gateway Agent [11]. The

[12] http://jaca-web.sourceforge.net

Webbridge Framework enables a seamless integration of the Jadex BDI agent framework [17] with JSP technology, combining the strength of agent-based computing with Web interactions. In particular, the framework extends the the Model 2 architecture – which brings the Model-View-Controller (MVC) pattern in the context of Web application development – to include also agents, replacing the controller with a bridge to an agent application, where agents react to user requests. JACK WebBot is a framework on top of the JACK BDI agent platform which supports the mapping of HTTP requests to JACK event handlers, and the generation of responses in the form of HTML pages. Using WebBot, you can implement a Web application which makes use of JACK agents to dynamically generate Web pages in response to user input. Finally, the JADE Gateway Agent is a simple interface to connect any Java non-agent application – including Web Applications based on Servlets and JSP – to an agent application running on the JADE platform [2].

All these approaches explore the use of agent technologies on the *server* side of Web Applications, while in our work we focus on the *client* side. So – roughly speaking – our agents are running *not* on a Web server but inside the Web browser, so in a fully decentralised fashion. Indeed, these two views can be combined together so as to frame an agent-based way to conceive next generation Web applications, with agents running on both the client and server side. Accordingly, this is a first important future work for JaCa-Web, extending the framework also on the server side. Indeed, further work is needed to identify then what to put on the client side and on the server side – i.e. which part of the agent application – by evaluating criteria such as portability, network/client/server load, etcetera.

Then, the use of agents to represent concurrent and interoperable computational entities already sets the stage for a possible evolution of Web client applications into *Semantic Web* applications [3]. From the very beginning [10], research activity on the Semantic Web has always dealt with *intelligent agents* capable of reasoning on machine-readable descriptions of Web resources, adapting their plans to the open Internet environment in order to reach a user-defined goal, and negotiating, collaborating, and interacting with each other during their activities. So, a main future work accounts for extending the JaCa-Web platform with Semantic Web technologies: to this purpose, existing works such as JASDL [12], will be main references.

6 Conclusion

In this paper we investigated the application of agent-oriented programming technologies to the development of future Internet applications based on the Web, introducing the JaCa-Web framework. Following the cloud computing perspective, these kinds of applications will play more and more an important role in the future, replacing in many cases desktop applications and becoming the standard de facto approach – along with service oriented architectures – to develop distributed systems in general. In this context, we believe that agent-orientation can play an important role both as enabling technology for realising smart applications and as a level of abstraction to uniformly and straightforwardly deal with the main complexities that the design and development of such applications imply. JaCa-Web then provides a first concrete framework to bridge the gap between the theory and the practice, allowing for experimenting the use of BDI agents – supported by proper artifact-based environments - for the development of Web client applications.

References

1. Agent Oriented Software Pty. JACK intelligent agents webbot manual (1999-2008), `http://www.aosgrp.com/documentation/jack/webbot_manual_web/index.html#thejackwebbotarchitecture`
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Chichester (2007)
3. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American (2001)
4. Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming: Languages, Platforms and Applications, vol. 1. Springer, Heidelberg (2005)
5. Rafael, B., Jomi, H., Mike, W.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd, Chichester (2007)
6. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications, vol. 2. Springer, Heidelberg (2009)
7. Fraternali, P., Rossi, G., Sanchez-Figueroa, F.: Rich Internet Applications. IEEE Internet Computing 14, 9–12 (2010)
8. Haller, P., Vetta, A.: Event-based programming without inversion of control. In: Lightfoot, D.E., Ren, X.-M. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (2006)
9. Haller, P., Vetta, A.: Actors that unify threads and events. In: Murphy, A.L., Ryan, M. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 171–190. Springer, Heidelberg (2007)
10. Hendler, J.: Agents and the Semantic Web. IEEE Intelligent Systems 16(2), 30–37 (2001)
11. JADE gateway agent (JADE 4.0 api) , `http://jade.tilab.com/doc/api/jade/wrapper/gateway/jadegateway.html`
12. Klapiscak, T., Bordini, R.H.: JASDL: A practical programming approach combining agent and semantic web technologies. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2008. LNCS (LNAI), vol. 5397, pp. 91–110. Springer, Heidelberg (2009)
13. Minotti, M., Piancastelli, G., Ricci, A.: An agent-based programming model for developing client-side concurrent web 2.0 applications. In: Filipe, J., Cordeiro, J. (eds.) Web Information Systems and Technologies. Lecture Notes in Business Information Processing, vol. 45. Springer, Heidelberg (2010)
14. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems 17 (3) (2008)
15. Pallis, G.: Cloud computing: The new frontier of internet computing. IEEE Internet Computing 14, 70–73 (2010)

16. Pokahr, A., Braubach, L.: The webbridge framework for building web-based agent applications. In: Dastani, M.M., El Fallah Seghrouchni, A., Leite, J., Torroni, P. (eds.) LADS 2007. LNCS (LNAI), vol. 5118, pp. 173–190. Springer, Heidelberg (2008), `http://dx.doi.org/10.1007/978-3-540-85058-8_11`
17. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In: Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F. (eds.) Multi-Agent Programming, Kluwer, Dordrecht (2005)
18. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
19. Ricci, A., Piunti, M., Acay, L.D., Bordini, R., Hübner, J., Dastani, M.: Integrating artifact-based environments with heterogeneous agent-programming platforms. In: Proceedings of 7th International Conference on Agents and Multi Agents Systems, AAMAS 2008 (2008)
20. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, vol. 2, pp. 259–288. Springer, Heidelberg (2009)
21. Ricci, A., Viroli, M., Omicini, A.: The A&A programming model and technology for developing agent environments in MAS. In: Dastani, M.M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 89–106. Springer, Heidelberg (2008)
22. Ricci, A., Viroli, M., Piancastelli, G.: simpA: An agent-oriented approach for programming concurrent applications on top of Java. Science of Computer Programming 76(1), 37–62 (2011)
23. Shoham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)

JaCa-Android: An Agent-Based Platform for Building Smart Mobile Applications

Andrea Santi, Marco Guidi, and Alessandro Ricci

DEIS, Alma Mater Studiorum – Università di Bologna
via Venezia 52, 47521 Cesena, Italy
{a.santi,a.ricci}@unibo.it,
marco.guidi7@studio.unibo.it

Abstract. Mobile applications are getting a strong momentum given the larger and larger diffusion of powerful mobile systems and related application platforms. A main example of such an application platform is given by Android, an open-source Java-based framework developed by Google for building and running applications on mobile devices.

On the other hand we do really believe that Agent-Oriented Programming (AOP) provides an effective level of abstraction for tackling the programming of mainstream software applications, in particular those that involve complexities related to concurrency, asynchronous events management and context-sensitive behaviour. Accordingly in this paper we support this claim in practice by discussing the use of a platform integrating two main agent programming technologies for the development of advanced mobile applications[1]. In detail this two technologies are: *(i)* *Jason* an agent programming language rooted on a strong notion of agency and *(ii)* CArtAgO environment programming framework. Here then we discuss the features of JaCa-Android, which makes it possible to exploit *Jason* and CArtAgO for straightforwardly programming smart applications on top of the Android platform using agent-based technologies.

1 Introduction

The value of Agent-Oriented Programming (AOP) [25] – including Multi-Agent Programming (MAP) – is often remarked and evaluated in the context of Artificial Intelligence (AI) and Distributed AI (DAI) problems. This is evident, for instance, by considering existing agent programming languages (see [5,7] for comprehensive surveys) – whose features are typically demonstrated by considering toy problems such as block worlds and alike.

Besides this view, we argue that the level of abstraction introduced by AOP is effective for organizing and programming software applications in general, starting from those programs that involve aspects related to reactivity, asynchronous interactions, concurrency, up to those involving different degrees of autonomy

[1] In literature it is also usual to refer to this kind of applications with the term *nomadic*: i.e. applications that follow the user in her everyday life.

and intelligence. Following this view, one of our current research lines investigates the adoption and the evaluation of existing agent-based programming languages and technologies for the development of applications in some of the most modern and relevant application domains. In this context, a relevant one is represented by next generation mobile applications. Applications of this kind are getting a strong momentum given the diffusion of mobile devices which are more and more powerful, in terms of computing power, memory, connectivity, sensors and so on. Main examples are smart-phones such as the iPhone and Android-based devices. On the one side, smart mobile applications share more and more features with desktop applications, and eventually extending such features with capabilities related to context-awareness, reactivity, usability, and so on, all aspects that are important in the context of Internet of Things and Ubiquitous Computing scenarios. All this increases – on the other side – the complexity required for their design and programming, introducing aspects that – we argue – are not suitably tackled by mainstream programming languages such as the object-oriented ones.

So, in this paper we discuss the application of an agent-oriented programming platform called JaCa for the development of smart mobile applications. Actually JaCa is not a new platform, but simply the integration of two existing agent programming technologies: *Jason* [6] agent programming language and platform, and CArtAgO [22] framework, for programming and running the environments where agents work. JaCa is meant to be a general-purpose programming platform, so useful for developing software applications in general. In order to apply JaCa to mobile computing, we developed a porting of the platform on top of Google Android, which we refer as JaCa-Android. Google Android is an open-source software stack for mobile devices provided by Google that includes an operating system (Linux-based), middleware, SDK and key applications.

The remainder of the paper is organised as follows: in Section 2 we provide a brief overview of the JaCa platform – which we consider part of the background of this paper; then, in Section 3 we introduce and discuss the application of JaCa for the development of smart mobile applications on top of Android, and in Section 4 we describe some practical application samples useful to evaluate the approach. Then in Section 5 we report the related works while in Section 6 we briefly discuss: *(i)* some open issues related to JaCa-Android and, more generally, to the use of current agent-oriented programming technologies for developing applications and *(ii)* related future works. Finally Section 7 concludes the paper.

2 Agent-Oriented Programming for Mainstream Application Development – The JaCa Approach

An application in JaCa is designed and programmed as a set of agents which work and cooperate inside a common environment. Programming the application means then programming the agents on the one side, encapsulating the logic of control of the tasks that must be executed, and the environment on the other side, as a first-class abstraction providing the actions and functionalities exploited by agents to do their tasks. It is worth remarking that this is an *endogenous* notion of environment, i.e. the environment here is part of the software system to be developed [23].

More specifically, in JaCa *Jason* [6] is adopted as programming language to implement and execute the agents and CArtAgO [22] as the framework to program and execute the environments.

Being a concrete implementation of an extended version of AgentSpeak(L) [20], *Jason* adopts a BDI (Belief-Desire-Intention)-based computational model and architecture to define the structure and behaviour of individual agents. In that, agents are implemented as reactive planning systems: they run continuously, reacting to events (e.g., perceived changes in the environment) by executing plans given by the programmer. Plans are courses of actions that agents commit to execute so as to achieve their goals. The pro-active behaviour of agents is possible through the notion of goals (desired states of the world) that are also part of the language in which plans are written. Besides interacting with the environment, *Jason* agents can communicate by means of speech acts.

On the environment side, CArtAgO – following the A&A meta-model [18,24] – adopts the notion of *artifact* as first-class abstraction to define the structure and behaviour of environments and the notion of *workspace* as a logical container of agents and artifacts. Artifacts explicitly represent the environment resources and tools that agents may dynamically instantiate, share and use, encapsulating functionalities designed by the environment programmer. In order to be used by the agents, each artifact provides a usage interface composed by a set of *operations* and *observable properties*. Operations correspond to the actions that the artifact makes it available to agents to interact with such a piece of the environment; observable properties define the observable state of the artifact, which is represented by a set of information items whose value (and value change) can be perceived by agents as percepts. Besides observable properties, the execution of operations can generate signals perceivable by agents as percepts, too. As a principle of composability, artifacts can be assembled together by a link mechanism, which allows for an artifact to execute operations over another artifact. CArtAgO provides a Java-based API to program the types of artifacts that can be instantiated and used by agents at runtime, and then an object-oriented data-model for defining the data structures used in actions, observable properties and events.

The notion of workspace is used to define the topology of complex environments, that can be organised as multiple sub-environments, possibly distributed over the network. By default, each workspace contains a predefined set of artifacts created at boot time, providing basic actions to manage the overall set of artifacts (for instance, to create, lookup, link together artifacts), to join multiple workspaces, to print messages on the console, and so on.

JaCa integrates *Jason* and CArtAgO so as to make the use of artifact-based environments by *Jason* agents seamless. To this purpose, first, the overall set of external actions that a *Jason* agent can perform is determined by the overall set of artifacts that are actually available in the workspaces where the agent is working.

So, the action repertoire is dynamic and can be changed by agents themselves by creating, disposing artifacts. Then, the overall set of percepts that a *Jason* agent can observe is given by the observable properties and observable events of the artifacts available in the workspace at runtime. Actually an agent can explicitly select which artifacts to observe, by means of a specific action called *focus*. By observing an artifact, artifacts' observable properties are directly mapped into beliefs in the belief-base, updated automatically each time the observable properties change their value. So a *Jason* agent can specify plans reacting to changes to beliefs that concern observable properties or can select plans according to the value of beliefs which refer to observable properties. Artifacts' signals instead are not mapped into the belief-base, but processed as non persistent percepts possibly triggering plans—like in the case of message receipt events. Finally, the *Jason* data-model – essentially based on Prolog terms – is extended to manage also (Java) objects, so as to work with data exchanged by performing actions and processing percepts.

A full description of *Jason* language/platform and CArtAgO framework – and their integration – is out of the scope of this paper: the interested reader can find details in literature [22,21] and on *Jason* and CArtAgO open-source web sites[2],[3].

3 Programming Smart Mobile Applications with JaCa

In this section we describe how JaCa's features can be effectively exploited to program smart mobile applications, providing benefits over existing non-agent approaches. First, we briefly sketch some of the complexities related to the design and programming of such a kind of applications; then, we describe how these are addressed in JaCa-Android—which is the porting of JaCa on Android, extended with a predefined set of artifacts specifically designed for exploiting Android functionalities.

3.1 Programming Mobile Applications: Complexities

Mobile systems and mobile applications have gained a lot of importance and magnitude both in research and industry over the last years. This is mainly due to the introduction of mobile devices such as the iPhone[4] and the most modern Android[5]-based devices that changed radically the concept of smart-phone thanks to: *(i)* hardware specifications that allow to compare these devices to miniaturised computers, situated – thanks to the use of practically every kind of known connectivity (GPS, WiFi, bluetooth, etc.) – in a computational network which is becoming more and more similar to the vision presented by both the Internet of Things and ubiquitous computing, and *(ii)* the evolution of both the smart-phone O.S. (Apple iOS, Android, Meego[6]) and their related SDK.

[2] `http://jason.sourceforge.net`
[3] `http://cartago.sourceforge.net`
[4] `http://www.apple.com/iphone/`
[5] `http://www.android.com/`
[6] `http://meego.com`

These innovations produce a twofold effect: on the one side, they open new perspectives, opportunities and application scenarios for these new mobile devices; on the other side, they introduce new challenges related to the development of the mobile applications, that are continuously increasing their complexity [2,15]. These applications – due to the introduction of new usage scenarios – must be able to address issues such as concurrency, asynchronous interactions with different kinds of services (Web sites/Services, social-networks, messaging/mail clients, etc.) and must also expose a user-centric behaviour governed by specific context information (geographical position of the device, presence/absence of different kinds of connectivity, events notification such as the reception of an e-mail, etc.).

To cope with these new requirements, Google has developed the Android SDK[7], which is an object-oriented Java-based framework meant to provide a set of useful abstractions for engineering mobile applications on top of Android mobile devices. Among the main coarse-grain components introduced by the framework to ease the application development we mention here:

– Activities: an activity provides a GUI for one focused endeavor the user can undertake. For example, an activity might present a list of menu items users can choose, list of contacts to send messages to, etc.
– Services: a service does not have a GUI and runs in the background for an indefinite period of time. For example, a service might play background music as the user attends to other matters.
– Broadcast Receiver: a broadcast receiver is a component that does nothing but receive and react to broadcast announcements. Many broadcasts originate in system code—for example, announcements that the timezone has changed, that the battery is low, etc.
– Content providers: a content provider makes a specific set of the application's data available to other applications. The data can be stored in the file system, in an SQLite database, etc.

In Android, interactions among components are managed using a messaging facility based on the concepts of Intent and IntentFilter. An application can request the execution of a particular operation – that could be offered by another application or component – simply providing to the O.S. an Intent properly characterised with the information related to that operation. So, for example, if an application needs to display a particular Web page, it expresses this need creating a proper Intent instance, and then sending this instance to the Android operating system. The O.S. will handle this request locating a proper component – e.g. a browser – able to manage that particular Intent. The set of Intents manageable by a component are defined by specifying, for that component, a proper set of IntentFilters.

Generally speaking, these components and the Intent-based interaction model are useful – indeed – to organise and structure applications; however – being the framework fully based on an object-oriented virtual machine and language

[7] `http://developer.android.com/sdk/index.html`

such as Java – they do not shield programmers from using callbacks, threads, and low-level synchronisation mechanisms as soon as applications with complex reactive behaviour are considered. For instance, in classic Android applications asynchronous interactions with external resources are still managed using polling loops or some sort of mailbox; context-dependent behaviour must be realised staining the source code with a multitude of if statements; concurrency issues must be addressed using Java low-level synchronisation mechanisms. So, more generally, we run into the problems that typically arise in mainstream programming languages when integrating one or multiple threads of control with the management of asynchronous events, typically done by callbacks.

In the next section we discuss how agent-oriented programming and, in particular, the JaCa programming model, are effective to tackle these issues at a higher-level of abstraction, making it possible to create more clear, terse and extensible programs.

3.2 An Agent-Oriented Approach Based on JaCa

By adopting the JaCa programming model, a mobile Android application can be realised as one or multiple workspaces in which *Jason* agents are used to encapsulate the logic and the control of tasks involved in the mobile application, and artifacts are used as tools for agents to seamlessly exploit available Android device/platform components and services.

From a conceptual viewpoint, this approach makes it possible to keep the same level of abstraction – based on agent-oriented concepts – both when designing the application and when concretely implementing it using *Jason* and CArtAgO. The design phase is rooted on the A&A meta-model [18,24] which allows to conceive an application: *(i)* defining its topology by introducing one or more workspaces, *(ii)* choosing the set of agents to include on the base of the application needs and *(iii)* by properly identifying the set of artifacts (i.e. tools and resources) to introduce in order to facilitate the agents' work. Then, during the implementation phase the developer can realise the application she has previously conceived using a set of first-class abstractions that directly refer the ones used in the design phase: agents are implemented in *Jason* while workspaces and artifacts can be realised using the CArtAgO framework. In this way we are able to provide developers a uniform guideline – without conceptual gaps between the abstractions used in the analysis and implementation phases – that drives the whole engineering process of a mobile application.

From a programming point of view, the agent paradigm makes it possible to tackle quite straightforwardly some of the main challenges mentioned in previous sections. In particular, the adoption of an agent-based programming language like *Jason* which promotes – being based on the BDI architecture – a strong notion of agency easily allows to address the following issues:

– Task and activity oriented behaviours can be directly mapped onto agents, possibly choosing different kinds of concurrent architectures according to the needs—either using multiple agents to concurrently execute tasks, or using a single agent to manage the interleaved execution of multiple tasks.

– Asynchronous interactions can be managed by properly specifying the agents' reactive behaviour in relation to the reception of particular percepts (e.g. the reception of a new e-mail).
– It is possible to realise applications that seamlessly integrate pro-active and reactive behavior thanks to the agents' control architecture. As explained above realise such an integration is an important programming issue [12,11]. In standard Android applications to make Android components aware of the occurrence of a certain event, it is necessary to register proper callbacks that encapsulate the source code that manages such event. This leads to: *(i)* inversion of control in programs and *(ii)* the proliferation of so-called *asynchronous spaghetti-code*. Using JaCa instead this issue can be easily solved: events generated by artifacts are automatically translated in percepts that *Jason* agents can observe and then use for autonomously choosing – in forthcoming reasoning cycles – the best actions to do in order to perform some kind of task.
– Agents' capability of adapting the behaviour on the basis of the current context information can be effectively exploited to realise context-sensitive and context-dependent applications.

It is worth remarking that we decided to adopt *Jason* as our reference agent programming language because it allows us to easily solve the issues just presented – mainly thanks to the adoption of the BDI architecture – and also because it is one of the most mature, most used, and most well documented existing open-source BDI-based agent programming languages. However other BDI-based agent programming languages or even other agent-based languages rooted on a strong notion of agency can be used as well.

A large body of MAS literature has remarked the key role that the notion of environment can play as first-class abstraction to design and develop MAS [27]. CArtAgO is a computational framework suited to play such a key role by providing to MAS developers another programming dimension, the environment dimension, for engineering a MAS system [22]. The environment dimension, and in particular CArtAgO, can play an important role also in the context of mobile applications, for example: *(i)* from an agent point of view environment resources (i.e. artifacts) can be used for bringing at the agent level proper tools that wrap any kind of functionality provided by the underlying mobile application platform hiding low level implementations details, and *(ii)* thanks to artifacts architecture model, it is provided a built-in support for the realization of proper coordination and synchronization mechanisms [19]—an important issue when distributed and concurrent applications are of concerns.

To see all this issues in practice, we developed a porting of JaCa on top of Android – referred as JaCa-Android – available as open-source project[8]. Fig. 1 shows an abstract representation of the levels characterising the JaCa-Android platform and of a generic application running on top of it.

The platform includes a set of predefined types of artifacts (`Broadcast ReceiverArtifact`, `ActivityArtifact`, `ServiceArtifact`, `ContentProvider`

[8] `http://jaca-android.sourceforge.net`

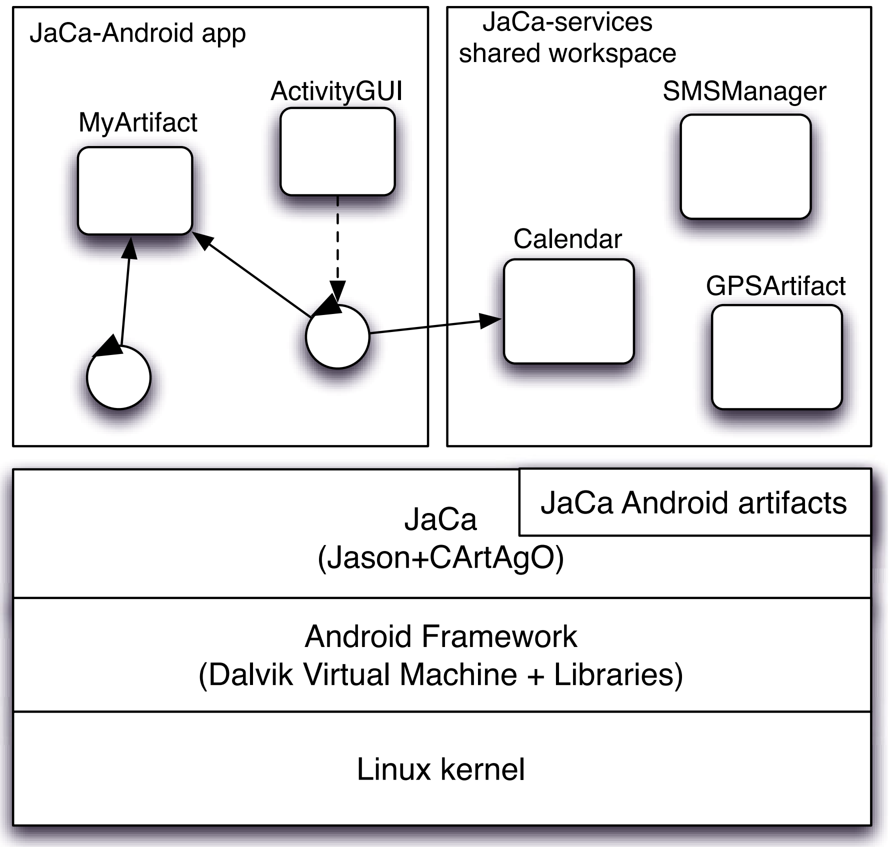


Fig. 1. Abstract representation of the JaCa-Android platform – with in evidence the different agent technologies on which the platform is based – and of a generic applications running on top of it

`Artifact`) specifically designed to build compliant Android components. So, standard Android components become fully-fledged artifacts that agents and agent developers can exploit without worrying and knowing about infrastructural issues related to the Android SDK. This makes it possible for developers to conceive and realise mobile applications that are seamlessly integrated with the Android SDK, possibly interacting/re-using every component and application developed using the standard SDK. This integration is fundamental in order to guarantee to developers the re-use of existing legacy – i.e. the standard Android components and applications – and for avoiding the development of the entire set of functionalities required by an application from scratch.

Besides, the platform also provides a set of artifacts that encapsulate some of the most common functionalities used in the context of smart mobile applications. In detail these artifacts are:

– `SMSManager/MailManager`, managing sms/mail-related functionalities (send and receive sms/mail, retrieve stored sms/mail, etc.).
– `GPSManager`, managing gps-related functionalities (e.g. geolocalisation of the device).
– `CallManager`, providing functionalities for handling – answer/reject – phone calls.
– `ConnectivityManager`, managing the access to the different kinds of connectivity supported by the mobile device.
– `CalendarArtifact`, providing functionalities for managing the built-in Google calendar.

These artifacts, being general purpose, are situated in a workspace called jaca-services (see Fig. 1) which is shared by all the JaCa-Android applications—being stored and executed into a proper Android service installed with the JaCa-Android platform. More generally, any JaCa-Android workspace can be shared among different applications—promoting, then, the modularisation and the reuse of the functionalities provided by JaCa-Android applications.

In next section we discuss more in detail the benefits of the JaCa programming model for implementing smart mobile applications by considering some application samples that have been developed on top of JaCa-Android.

4 Evaluation through Practical Examples

Just to have a taste of the approach a first simple example is provided. Table 1 shows a snippet of an agent playing the role of smart user assistant, with the task of managing the notifications related to the reception of SMS messages: as soon as an SMS is received, a notification must be shown to the user. The `SMSManager` artifact described above is used to manage SMS messages, in particular this artifact generates an observable event `sms_received` each time a new SMS is received. A `ViewerArtifact` artifact is used to show SMS messages on the screen and to keep track – by means of the `state` observable property – of the current status of the viewer, that is if it is currently visualised by the user on the smart-phone screen or not. Finally, a `StatusBarArtifact` artifact is used instead to show messages on the Android status bar, providing a `showNotification` operation to this end. Depending on what the user is actually doing and visualising, the agent shows the notification in different ways. The behavior of the agent, once completed the initialization phase (lines 00-04), is governed by two reactive plans. The first one (lines 6-10) is applicable when a new message arrives and the `ViewerArtifact` is not currently visualised on the smart-phone's screen. In this case, the agent performs a `showNotification` action to notify the user of the arrival of a new message using the status bar (Fig. 2, *(a)*). The second plan instead (lines 12-14) is applicable when the `ViewerArtifact` is currently displayed on

Table 1. Source code of the *Jason* agent that manages the SMS notifications

```
00  !init.
01
02  +!init
03  <-  focus("SMSManager");
04      focus("ViewerArtifact").
05
06  +sms_received(Source, Message)
07    : not (state("running") & session(Source))
08    <-  showNotification("jaca.android:drawable/notification",
09           Source, Message, "jaca.android.sms.SmsViewer", Id);
10        +session(Source, Id).
11
12  +sms_received(Source, Message)
13    : state("running") & session(Source)
14    <-  append(Source, Message).
```

screen and therefore the agent could notify the SMS arrival by simply appending the SMS to the received message list showed by the viewer (Fig. 2, *(b)*): this is done by executing the `append` operation provided by `ViewerArtifact`.

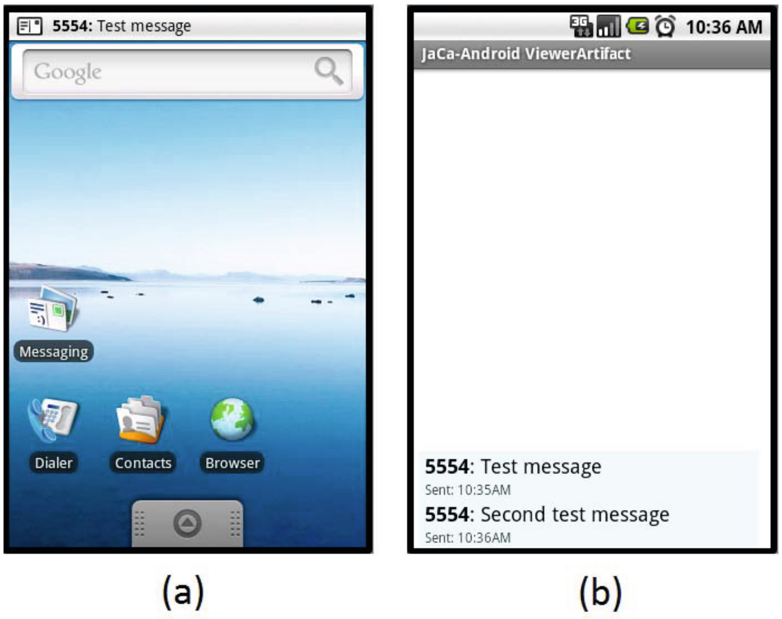
From the example, it should be clear that for a developer able to program using the JaCa programming model realise a JaCa-Android application is a quite straightforward experience. Indeed, following the JaCa approach, she can continue to engineer the business logic of the applications by suitably defining the *Jason* agent's behavior, and it only need to acquire the ability to work with the artifacts that are specific of the mobile application context.

The second example starts to be more significant, and it aims to show how the approach allows to easily realise context-sensitive mobile applications. For this purpose, we consider a JaCa-Android application inspired to Locale[9], one of the most famous Android applications and also one of the winners of the first Android Developer Contest[10]. This application (JaCa-Locale) can be considered as a sort of intelligent smart-phone manager realised using a simple *Jason* agent. The agent during its execution uses some of the built-in JaCa-Android artifacts described in Section 3.2 and two application-specific artifacts: a `PhoneSettingsManager` artifact used for managing the device ringtone/vibration and the `ContactManager` used for managing the list of contacts stored into the smart-phone (this list is an observable property of the artifact, so directly mapped into agents beliefs). The agent manages the smart-phone behaviour discriminating the execution of its plans on the basis of a comparison among its actual context information and a set of user preferences that are specified into the agent's plans contexts. Table 2 reports a snippet of the *Jason* agent used in JaCa-Locale, in particular the plans shown in Table 2 are the ones responsible of the context-dependent management of the incoming phone calls.

The behaviour of the agent, once completed the initialisation phase (lines 00-07), is governed by a set of reactive plans. The first two plans (lines 9-15) are used for regulating the ringtone level and the vibration for the incoming calls on the basis of the notifications provided by the `CalandarArtifact` about

[9] http://www.twofortyfouram.com/

[10] http://code.google.com/intl/it-IT/android/adc/



5554: Test message
Sent: 10:35AM
5554: Second test message
Sent: 10:36AM

Fig. 2. The two different kinds of SMS notifications: (*a*) notification performed using the standard Android status bar, and (*b*) notification performed using the `ViewerArtifact`

the start or the end of an event stored into the user calendar. Instead, the behaviour related to the handling of the incoming calls is managed by the two reactive plans `incoming_call` (lines 17-28). The first one (lines 17-19) is applicable when a new incoming call arrives and the phone owner is not busy, or when the incoming call is considered critical. In this case the agent normally handles the incoming call – the ringtone/vibration settings have already been regulated by the plans at lines 9-15 – using the `handle_call` operation provided by the `CallManager` artifact. The second plan instead (lines 21-28) is applicable when the user is busy and the call does not come from a relevant contact. In this case the phone call is automatically rejected using the `drop_call` operation of the `CallManager` artifact (line 24), and an auto-reply message containing the motivation of the user unavailability is sent back to the contact that performed the call. This notification is sent – using one of the `handle_auto_reply` plans (lines 30-34) – via sms or via mail (using respectively the `SMSManager` or the `MailManager`) depending on the current availability of the WiFi connection on the mobile device (availability checked using the `wifi_status` observable property of the `ConnectivityManager`). It is worth remarking that `busy` and `is_call_critical` refer to rules – not reported in the source code – used for checking respectively: *(i)* if the phone owner is busy – by checking the belief related to one of the `CalendarArtifact` observable properties

Table 2. Source code snippet of the JaCa-Locale *Jason* agent

```
!init.

+!init
  <- focus("SMSManager"); focus("MailManager");
     focus("CallManager"); focus("ContactManager");
     focus("CalendarArtifact");
     focus("PhoneSettingsManager");
     focus("ConnectivityManager").

+cal_event_start(EventInfo) : true
  <- set_ringtone_volume(0);
     set_vibration(on).

+cal_event_end(EventInfo) : true
  <- set_ringtone_volume(100);
     set_vibration(off).

+incoming_call(Contact, TimeStamp)
  : not busy(TimeStamp) | is_call_critical(Contact)
  <- handle_call.

+incoming_call(Contact, TimeStamp)
  : busy(TimeStamp) & not is_call_critical(Contact)
  <- get_event_description(TimeStamp,EventDescription);
     drop_call;
     .concat("Sorry, I'm busy due
       to", EventDescription, "I will call you back
       as soon as possible.", OutStr);
     !handle_auto_reply(OutStr).

+!handle_auto_reply(Reason) : wifi_status(on)
  <- send_mail("Auto-reply", Reason).

+!handle_auto_reply(Reason): wifi_status(off)
  <- send_sms(Reason).
```

(`current_app`) – or *(ii)* if the call is critical – by checking if the call comes from one of the contact in the `ContactManager` list considered critical: e.g. the user boss/wife.

Generalising the example, context-sensitive applications can be designed and programmed in terms of one or more agents with proper plans that are executed only when the specific context conditions hold.

The example is useful also for highlighting the benefits introduced by artifact-based endogenous environments: *(i)* it makes it possible to represent and exploit platform/device functionalities at an agent level of abstractions – so in terms of actions and perceptions, modularised into artifacts; *(ii)* it provides a strong *separation of concerns*, in that developers can fully separate the code that defines the control logic of the application (into agents) from the reusable functionalities (embedded into artifacts) that are need by the application, making agents' source code more dry.

The third application sample – called SmartNavigator (see Fig. 3 for a screenshot) – aims at showing the effectiveness of the approach in managing asynchronous interactions with external resources, such as – for example – Web Services. This application is a sort of smart navigator able to assist the user during her trips in an intelligent way, taking into account the current traffic conditions.

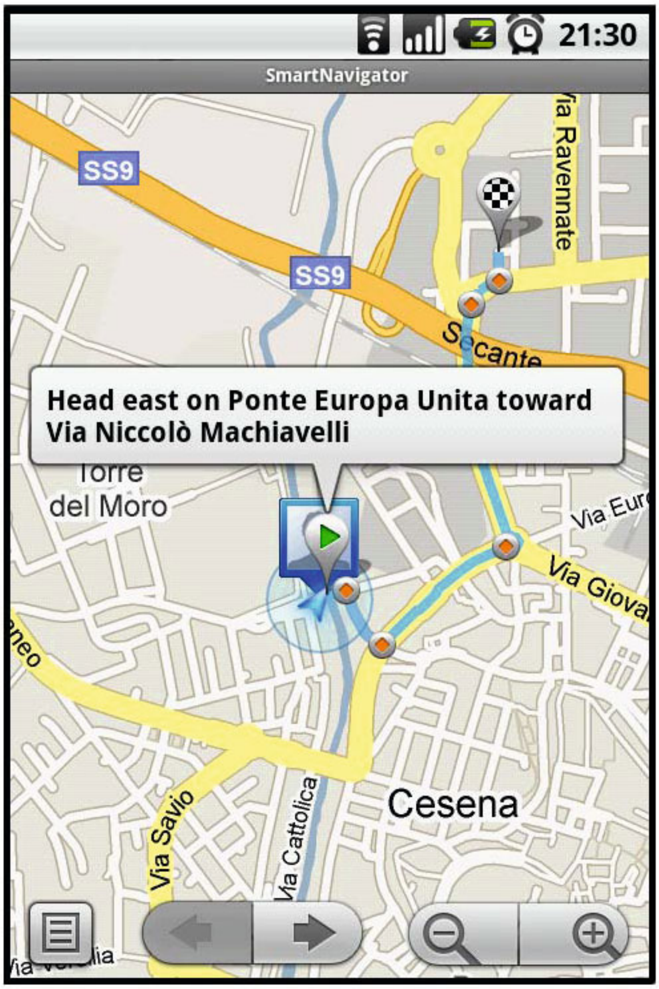


Fig. 3. Screenshot of the SmartNavigator application that integrate in its GUI some of the Google Maps component s for showing: *(i)* the user current position, *(ii)* the road directions, and *(iii)* the route to the designed destination

The application is realised using a single *Jason* agent and four different artifacts: *(i)* the `GPSManager` used for the smart-phone geolocalisation, *(ii)* the `GoogleMapsArtifact`, an artifact specifically developed for this application, used for encapsulating the functionalities provided by Google Maps (e.g. calculate a route, show points of interest on a map, etc.), *(iii)* the `SmartNavigatorGUI`, an artifact developed on the basis of the `ActivityArtifact` and some other Google Maps components, used for realizing the GUI of the application and *(iv)* an artifact, `TrafficConditionsNotifier`, used for managing the interactions with a Web site[11] that provides real-time traffic information.

Table 3 shows a snippet of the agent source code. The agent main goal `assist_user_trips` is managed by a set of reactive plans that are structured in a hierarchy of sub-goals – handled by a set of proper sub-plans. The agent has a set of initial beliefs (lines 00-01) and an initial plan (lines 5-9) that manages the initialisation of the artifacts that will be used by the agent during its execution. The first plan, reported at lines 11-12, is executed after the reception of an event related to the modification of the `SmartNavigatorGUI` `route` observable property – a property that contains both the starting and arriving locations provided in input by the user. The handling of this event is managed by the `handle_navigation` plan that: *(i)* retrieves (line 15) and updates the appropriate agent beliefs (line 16 and 19), *(ii)* computes the route using an operation provided by the `GoogleMapsArtifact` (`calculate_route` lines 17-18), *(iii)* makes the subscription – for the route of interest – to the Web site that provides the traffic information using the `TrafficConditionsNotifier` (lines 20-21), and finally *(iv)* updates the map showed by the application (using the `SmartNavigatorGUI` operations `set_current_position` and `update_map`, lines 22-23) with both the current position of the mobile device (provided by the observable property `current_position` of the `GPSManager`) and the new route.

In the case that no meaningful changes occur in the traffic conditions and the user strictly follows the indications provided by the SmartNavigator, the map displayed in the application GUI will be updated, until arriving to the designed destination, simply moving the current position of the mobile device using the plan reported at lines 34-38. This plan, activated by a change of the observable property `current_position`, simply considers (using the sub-plan `check_position_consistency` instantiated at line 36, not reported for simplicity) if the new device position is consistent with the current route (retrieved from the agent beliefs at line 35) before updating the map with the new geolocation information (line 37-38). In the case in which the new position is not consistent – i.e. the user chose the wrong direction – the sub-plan `check_position_consistency` fails. This fail is handled by a proper *Jason* failure handling plan (lines 40-42) that simply re-instantiate the `handle_navigation` plan for computing a new route able to bring the user to the desired destination from her current position (that was not considered in the previous route).

Finally, the `new_traffic_info` plan (lines 25-32) is worth of particular attention. This is the reactive plan that manages the reception of the updates related to the traffic conditions. If the new information are considered relevant with respect to the user preferences (sub-plan `check_info_relevance` instantiated at line 28 and not shown) then, on the basis of this information, the current route (sub-plan `update_route` instantiated at lines 29-30), the Web site subscription (sub-plan `update_subscription` instantiated at line 31), and finally the map displayed on the GUI (line 32) are updated.

[11] http://www.stradeanas.it/traffico/

Table 3. Source code snippet of the SmartNavigator Jason agent

```
preferences([...]).
relevance_range(10).

!assist_user_trips.

+!assist_user_trips
  <- focus("GPSManager");
     focus("GoogleMapsArtifact");
     focus("SmartNavigatorGUI");
     focus("TrafficConditionsNotifier").

+route(StartLocation, EndLocation)
  <- !handle_navigation(StartLocation, EndLocation).

+!handle_navigation(StartLocation, EndLocation)
  <- ?relevance_range(Range); ?current_position(Pos);
     -+leaving(StartLocation);-+arriving(EndLocation);
     calculate_route(StartLocation,
       EndLocation, OutputRoute);
     -+route(OutputRoute);
     subscribe_for_traffic_condition(OutputRoute,
       Range);
     set_current_position(Pos);
     update_map.

+new_traffic_info(TrafficInfo)
  <- ?preferences(Preferences);
     ?leaving(StartLocation); ?arriving(EndLocation);
     !check_info_relevance(TraffincInfo,Preferences);
     !update_route(StartLocation, EndLocation,
       TrafficInfo, NewRoute);
     !update_subscription(NewRoute);
     update_map.

+current_position(Pos)
  <- ?route(Route);
     !check_position_consistency(Pos, Route);
     set_current_position(Pos);
     update_map.

-!check_position_consistency(Pos, Route)
  : arriving(EndLocation)
  <- !handle_navigation(Pos, EndLocation).
```

So, this example shows how it is possible to seamlessly integrate the reactive behaviour of a JaCa-Android application – in this example the asynchronous reception of information from a certain source – with its pro-active behaviour—assisting the user during her trips. This integration allows to easily modify and adapt the pro-active behaviour of an application after the reception of new events that can be handled by proper reactive plans: in this example, the reception of the traffic updates can lead the SmartNavigator to consider a new route for the trip on the basis of the new information.

5 Related Works

Other works in literature discuss the use of agent-based technology on mobile devices. A good part of these works is not so recent and mainly considers the issue of porting agent technologies on limited capability devices and platforms—the only ones existing at that particular time. Noteworthy examples include AgentFactory Mirco Edition [17], 3APL-M [14], JADE [4]. A good survey of this kind of works can be found here [9]. Differently from these works, in this paper our target is the new generation of smartphones produced by the industry. Indeed, as described in Section 3.1, in the last few years the mobile scenario is drastically changed. So, a big part of the problems faced in these works – e.g. low computing power, very low memory availability – are no more issues, or at least are not so constraining when engineering an agent-based platform for the new generation of mobile devices.

Besides facing the porting of JaCa in the Android context, in our work a key issue concerns the investigation of the advantages brought by the adoption of agent-oriented programming level of abstraction for the development of complex mobile applications. Indeed our claim is that the agent paradigm has to offer several benefits for the development of these applications. For this purpose we aim at the developing of mature programming models and platforms – starting from JaCa and JaCa-Android – wherewith realise agent-based applications for supporting our claim in practice.

A part of more recent literature describes the porting of agent-based models and technologies to the Android platform. In [26] is presented a new specialisation of the Jade-Leap platform for the Android context. This article discusses the port of agent-based technologies to the Android world but the proposed solution is still centered on the problematics related to old mobile devices—so again devices with low computing capabilities. The authors promote an approach in which a Jade container is split in a back-end – typically residing on a desktop machine – and a front-end – where the mobile application actually lives. The agent-based application in the front-end communicates with its back-end in order to delegate a big part of the computational workload related to the business logic of the application. We argue that, due to current specifications of mobile devices, this approach is over-constraining and restrictive. Indeed now it is possible to conceive an agent-based mobile platform that exploits the full power of the new generation of devices: this is the approach that we promote in this article. Finally [1] discusses how the authors' agent-model could be implemented on top of the Android SDK. The porting issue assumes a key role also in this paper at the expense of the more visionary aspects related to the impacts of adopting agent-based technologies in the mobile context—which is instead a core part of this paper.

6 Open Issues and Future Work

JaCa-Android is just a prototype, however even as it is it allows to realise applications that are quite fast and responsive. Indeed, during our firsts experiments and tests on the platform, also concerning the applications described in this article, we have noticed no particular problems. JaCa-Android applications run quite smoothly, without experiencing lags or latency issues, and their execution do not compromise the execution of other applications on the device (so JaCa-Android applications do not use too much CPU). These are just the firsts experimental results and they relevance is merely qualitative, however they represent:

– A good starting point for future platform improvements.
– A first positive feedback for both the real usability of JaCa-Android as a platform for developing mobile application, and for future comparisons with the classical Android development platform.

As remarked before JaCa-Android is still in early development phases and it needs further developments for stressing more in depth the benefits related to the adoption of agent-oriented programming for realising new generation mobile applications, especially in comparisons with mainstream non-agent platforms. Therefore, in future works we aim at improving JaCa-Android in order to tackle some other important features of modern mobile applications such as the smart use of the battery and the efficient management of the computational workload of the device. These improvements are indeed fundamentals in order to promote further analysis and comparisons between applications realised with JaCa-Android and the standard Android platform:

– Which are the differences in performances between the two platforms?
– How many different applications can be concurrently executed in each platform?
– Which are the differences in battery consumption and CPU-use?

These are just a part of the questions that we would like to answer in future works.

Besides the advantages described in previous section, the application of current agent programming technologies to the development of concrete software systems such as mobile applications have been useful to focus some main weaknesses that these technologies currently have to this end. Here we have identified three main general issues that will be subject of future work:

(i) Devising of a notion of type *for agents and artifacts*—current agent programming languages and technologies lack of a notion of type as the one found in mainstream programming languages and this makes the development of large system hard and error-prone. This would make it possible to detect many errors at compile time, allowing for strongly reducing the development time and enhancing the safety of the developed system. In JaCa we have a notion of type just for artifacts: however it is based on the lower OO layer and so not expressive enough to characterise at a proper level of abstraction the features of environment programming.

(ii) Improving modularity *in agent definition*—this is a main issue already recognised in the literature [8,10,13], where constructs such as *capabilities* have been proposed to this end. *Jason* still lacks of a construct to properly modularise and structure the set of plans defining an agent's behaviour—a recent proposal is described here [16].

(iii) Improving the integration with the OO layer – To represent data structures, *Jason* – as well as the majority of agent programming languages – adopts

Prolog terms, which are very effective to support mechanisms such as unification, but quite weak – from an abstraction and expressiveness point of view – to deal with complex data structures. Main agent frameworks (not languages) in Agent-Oriented Software Engineering contexts – such as Jade [3] or JACK[12] – adopt object-oriented data models, typically exploiting the one of existing OO languages (such as Java). By integrating *Jason* with CArtAgO, we introduced a first support to work with an object-oriented data model, in particular to access and create objects that are exchanged as parameters in actions/percepts. However, it is just a first integration level and some important aspects – such as the use of unification with object-oriented data structures – are still not tackled.

7 Conclusion

To conclude, we believe that agent-oriented programming – including multi-agent programming – would provide a suitable level of abstraction for tackling the development of complex software applications, extending traditional programming paradigms such as the Object-Oriented to deal with aspects such as concurrency, reactiveness, asynchronous interaction managements, dynamism and so on. In this paper, in particular, we showed the advantages of applying such an approach to the development of smart mobile applications on the Google Android platform, exploiting the JaCa-Android integrated platform. However, we argue that in order to stress and investigate the full value of the agent-oriented approach to this end, further works are needed. These future works should concern on the one side extensions and improvements of the proposed platform, and on the other side extensions of current agent languages and technologies – or the realisation of new ones – tackling main aspects that have not been considered so far, being not related to AI but to the principles of software development. This is the core of our current and future work.

References

1. Agüero, J., Rebollo, M., Carrascosa, C., Julián, V.: Does Android Dream with Intelligent Agents? In: International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008), pp. 194–204. Springer, Heidelberg (2008)
2. Battestini, A., Rosso, C.D., Flanagan, A., Miettinen, M.: Creating next generation applications and services for mobile devices: Challenges and opportunities. In: EEE 18th Int. Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC), pp. 1–4, 3-7 (2007)
3. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley, Chichester (2007)
4. Berger, M., Rusitschka, S., Toropov, D., Watzke, M., Schlichte, M.: Porting distributed agent-middleware to small mobile devices. In: AAMAS Workshop on Ubiquitous Agents on Embedded, Wearable and Mobile Devices (2002)
5. Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming: Languages, Platforms and Applications, vol. 1. Springer, Heidelberg (2005)

[12] `http://www.agent-software.com`

6. Bordini, R., Hübner, J., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak Using Jason. John Wiley & Sons, Ltd, Chichester (2007)
7. Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A.: Multi-Agent Programming: Languages, Platforms and Applications, vol. 2. Springer, Heidelberg (2009)
8. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible BDI agent modularization. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862, pp. 139–155. Springer, Heidelberg (2006)
9. Carabelea, C., Boissier, O.: Multi-agent platforms on smart devices: Dream or reality? In: Proceedings of the Smart Objects Conference (SOC 2003), Grenoble, France, pp. 126–129 (2003)
10. Dastani, M., Mol, C., Steunebrink, B.: Modularity in agent programming languages: An illustration in extended 2APL. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 139–152. Springer, Heidelberg (2008)
11. Haller, P., Odersky, M.: Event-based programming without inversion of control. Modular Programming Languages, 4–22 (2006)
12. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
13. Hindriks, K.: Modules as policy-based intentions: Modular agent programming in GOAL. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 156–171. Springer, Heidelberg (2008)
14. Koch, F., Meyer, J.-J.C., Dignum, F.P.M., Rahwan, I.: Programming deliberative agents for mobile services: The 3APL-M platform. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862, pp. 222–235. Springer, Heidelberg (2006)
15. König-Ries, B.: Challenges in mobile application development. IT - Information Technology 51(2), 69–71 (2009)
16. Madden, N., Logan, B.: Modularity and compositionality in jason. In: Braubach, L., Briot, J.-P., Thangarajah, J. (eds.) ProMAS 2009. LNCS, vol. 5919, pp. 237–253. Springer, Heidelberg (2010)
17. Muldoon, C., O'Hare, G.M.P., Collier, R.W., O'Grady, M.J.: Agent factory micro edition: A framework for ambient applications. In: Int. Conference on Computational Science, vol. (3), pp. 727–734 (2006)
18. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems 17 (3) (2008)
19. Omicini, A., Ricci, A., Viroli, M., Castelfranchi, C., Tummolini, L.: Coordination artifacts: Environment-based coordination for intelligent agents. In: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems AAMAS 2004, pp. 286–293. IEEE, Los Alamitos (2005)
20. Rao, A.S.: AgentSpeak(l): BDI agents speak out in a logical computable language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
21. Ricci, A., Piunti, M., Acay, L.D., Bordini, R., Hübner, J., Dastani, M.: Integrating artifact-based environments with heterogeneous agent-programming platforms. In: Proceedings of 7th International Conference on Agents and Multi Agents Systems, AAMAS 2008 (2008)
22. Ricci, A., Piunti, M., Viroli, M., Omicini, A.: Environment programming in CArtAgO. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms and Applications, vol. 2, pp. 259–288. Springer, Heidelberg (2009)

23. Ricci, A., Santi, A., Piunti, M.: Action and perception in multi-agent programming languages: From exogenous to endogenous environments. In: Proceedings of the Int. Workshop on Programming Multi-Agent Systems (ProMAS 2010), Toronto, Canada (2010)
24. Ricci, A., Viroli, M., Omicini, A.: The A&A programming model & technology for developing agent environments in MAS. In: Dastani, M., Seghrouchni, A.E.F., Ricci, A., Winikoff, M. (eds.) Programming Multi-Agent Systems. LNCS (LNAI), vol. 4908, pp. 91–109. Springer, Heidelberg (2007)
25. Soham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)
26. Ughetti, M., Trucco, T., Gotta, D.: Development of agent-based, peer-to-peer mobile applications on ANDROID with JADE. In: The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, UBICOMM 2008, pp. 287–294. IEEE, Los Alamitos (2008)
27. Weyns, D., Omicini, A., Odell, J.J.: Environment as a first-class abstraction in multi-agent systems. Autonomous Agents and Multi-Agent Systems 4(1), 5–30 (2007); Special Issue on Environments for Multi-agent Systems

An Alternative Approach for Reasoning about the Goal-Plan Tree Problem

Patricia Shaw[1] and Rafael H. Bordini[2]

[1] University of Durham, UK
p.h.shaw@durham.ac.uk
[2] Federal University of Rio Grande do Sul, Brazil
r.bordini@inf.ufrgs.br

Abstract. Agents programmed in BDI-inspired languages have goals to achieve and a library of plans that can be used to achieve them, typically requiring further goals to be adopted. This is most naturally represented by a structure that has been called a Goal-Plan Tree. One of the uses of such structure is in agent deliberation (in particular, deciding whether to commit to achieving a certain goal or not). In previous work, a Petri net based approach for reasoning about goal-plan trees was defined. This paper presents a constraint-based approach to perform the same reasoning, which is then compared with the Petri net approach.

Keywords: Agent Reasoning, Constraints, Goal-Plan Tree.

1 Introduction

Agents programmed in BDI-inspired languages have goals to achieve and a library of plans that can be used to achieve them, typically requiring further goals to be adopted. This is most naturally represented by a structure that has been called a Goal-Plan Tree. Whilst no planning takes place in such agents, a certain type of reasoning – done over such representation of agents' commitments towards goals to be achieved and the known courses of actions to achieve them – can significantly impact the agent's performance by judicious scheduling of the plan execution. More importantly, it can significantly improve *deliberation*, in the sense that an agent can make reasoned choices on whether to commit to achieving a new goal or not.

In the work by Thangarajah *et al.* [8,9,10], a *goal-plan tree* is used to represent the structure of the various plans and subgoals related to each goal for an individual agent. At each node of the tree, *summary information* is used to represent the various constraints under consideration. This is similar to previous work by Clement and Durfee [1,2,3], using summary information with Hierarchical Task Network (HTN) planning to co-ordinate the actions of multiple agents.

When using summary information, the amount of summary information to handle could potentially grow exponentially with the size of the goal-plan tree [3], which could have a significant impact on the performance of the agent for larger problems. A different approach was introduced by Shaw and Bordini [5], where a goal-plan tree is mapped into a Petri net in such a way as to avoid the need for summary information.

The work in [5] considered reasoning about both positive and negative effects of a plan on other plans using a Petri net based technique, while in [6] the focus is on reasoning about resources using Petri nets, which are then *combined* into a coherent reasoning process encompassing the reasoning about positive and negative interactions from [5]. These were evaluated based on an abstract scenario as well as a more concrete scenario using a simplified mars rover scenario.

In this paper, we present an alternative specific implementation of an approach to reasoning about positive, negative and resource interactions using a constraint logic programming approach developed in GNU Prolog to define a set of constraints that are eventually solved to generate a successful execution ordering of the plans to achieve an agent's goals. These are evaluated against the Petri net model using a common abstract scenario. While the approach described here is based on a specific implementation, the concepts and processes could be reapplied in other constraint (logic) programming or even constraint optimisation settings. However, the aim of this paper is to present an approach to solving a problem and experimentally compare it to another approach in order to identify situations where it may be preferable to apply one approach over the other.

The remainder of the paper is organised as follows. Section 2 shows the constraint-based approach with each of the three forms of reasoning incorporated. Section 3 shows the experimental results and analysis of the comparison of that approach to the Petri net approach for reasoning about the goal-plan tree problem. Section 4 concludes the paper.

2 Constraint-Based Approach

2.1 Goal-Plan Trees

A goal-plan tree consists of a top-level goal at the root, with one or more plans available to achieve that goal. Each of these plans may themselves include further subgoals forming the next level in the tree, followed by additional plans to achieve these subgoals[1]. All subgoals for a plan must be achieved for a plan to be successful, while only one plan option needs to be executed for a goal or subgoal to be successful. The simplest plans at the leaves of the tree will just contain a sequence of actions and no further subgoals. An example of a goal-plan tree is shown in Figure 1, which shows the goal-plan tree representation of a goal for a Mars Rover to collect a soil sample from a location then transmit the results back to Earth via the base station.

An agent will most likely have multiple top-level goals to achieve, each with its own goal-plan tree. While it is often straightforward for these to be achieved in sequence, it may be possible for the agent to achieve better performance by

[1] The term subgoals will always be used when referring to subgoals, while top-level goals will either be referred to as goals or top-level goals.

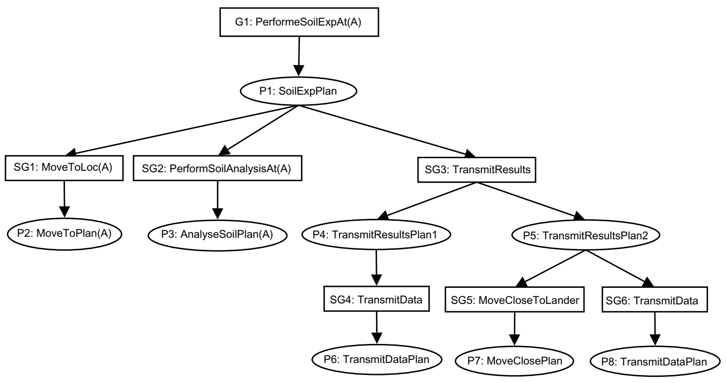


Fig. 1. Goal-plan tree for a Mars rover as used by Thangarajah *et al.* The goals and subgoals are represented by rectangles while the plans are represented by ovals.

attempting to achieve them in parallel. This can of course lead to problems where the goals interfere with each other and where resources are limited, so reasoning about that can help an agent succeed in achieving its goals and do so more efficiently. The three types of reasoning considered here are based on: (*i*) the limited availability of consumable resources, (*ii*) the potential for positive interactions between goals, and (*iii*) the risk of negative interference between goals, which are discussed in more detail in the following sections.

The approach developed here for reasoning about goals applies constraint satisfaction to find a solution to instances of the goal-plan tree problem. While the Petri net approach applied in [6] provided a natural representation of an agent's goal-plan tree into which the reasoning could be added, this approach provides a natural representation of the constraints to be handled by the agent in the form of resource constraints and interaction constraints. The constraints are represented using GNU Prolog[2].

2.2 Modelling a Goal-Plan Tree

The idea surrounding the model used for representing the goal-plan tree reasoning problem as a set of constraints is to find an ordering of the plans for all of the goals such that all the goals adopted are achieved and as many goals as possible are adopted.

To start with, the plans and goals are both defined as facts using a Prolog functor `node`, with the plans being represented by 5-tuples $\langle Pl, S, Pr, E, R\rangle$ where Pl is a unique identifier for each plan; S is the list of subgoals for achieving the plan; Pr is a list of preconditions and E is a list of effects caused by the plan; R is a list of pairs showing the resource requirements for the different resources that a plan uses. The plans at the bottom of the goal-plan tree that

[2] The shorthand notation `V1/V2` is used in GNU Prolog to represent pairs of values.

form the leaves of the tree will not have any subgoals listed in S, and not all plans will have preconditions, effects or resource requirements.

A series of "variables" is used to represent resources and the effects on the environment. The representation of available resources make use of dynamic facts that can be updated as the resources are consumed, (e.g. `resource(r1,50)`). In the plan definitions, the preconditions, effects and resources are all represented as pairs of values, for example `r1/5` represents the requirement of 5 units of resource $r1$. The preconditions and effects referring to various properties of the environment that can be modified are represented in a similar way with effect `e1/7` stating that the plan changes the variable representing the environment property $e1$ so that it has the value 7. Sections 2.4 and 2.5 describe how these are used to identify plans that can either be safely "merged" or that could interfere (thus needing to be scheduled accordingly).

Goals and subgoals require less details, so they are simply represented as pairs $\langle G, P\rangle$, where G is a unique identifier for the goal or subgoal and P is a non-empty list of plans that can be used to achieve G. The following Prolog sample from a goal definition shows a top-level goal node and a plan node that achieves this goal, itself using 1 unit of resource r1 and causing the effect of assigning the value 7 to variable e3, while having no preconditions required for it to start.

```
node(g1,[p1]).                              % Goal node
node(p1,[sg2,sg3],[],[e3/7],[r1/1]).   % Plan node
```

In order to reason about the tree structure, various predicates are defined to help query a goal-plan tree representation. These include listing all the plans in the sub-tree of a goal or plan, finding all the plan options for achieving a goal or subgoal, and querying the plan hierarchy within the goal-plan tree.

Where there is a choice of plans to achieve a goal or subgoal, only one of these needs to be used in order for the goal to be successful. The surplus plans can therefore be dropped from consideration, reducing the number of plans that need to be considered later on. When the plan being dropped contains subgoals, these are also removed from consideration. This is illustrated in Figure 2, where the plan and its sub-tree inside the dashed line is being dropped in preference of the alternative plan for achieving the subgoal.

In Prolog, this is defined as a series of predicates to "strip" the tree of the branch options:

```
branchOptions:-
    findall(O,option(_,O),All),
    branchStrip(All).
```

The clause above uses the `option(Goal,OptionList)` predicate to generate a list of all the sets of options for subgoal branches. `O` is a list of plans from which just one plan needs to be selected, so the variable `All`, the result of the `findall`, equates to a list of plan lists. Each of these lists of plans then needs to be considered, selecting one plan to keep and the remainder to disregard. By default, the plan that is kept is the first plan in the list. However, when resource

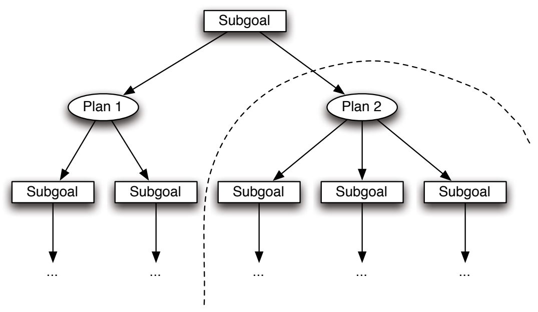


Fig. 2. Removal of surplus sub-trees where there is a choice of plans

reasoning is incorporated, the summary resource requirements for each branch is considered so the plan with the lowest summary resource requirements is kept.

```
branchStrip([]).
branchStrip([[H | T2] | T]):-
    rmBranch(T2),
    branchStrip(T).

rmBranch([]).
rmBranch([P|T]):-
    strip(P),
    rmBranch(T).
```

When removing plans, it is important to remember to remove the sub-tree formed from any subgoals that were required by the plan. This is handled by a final recursive predicate to iterate through the list ensuring each of the members of the sub-tree are removed. As it is possible for plans within the subtree of an optional plan to also contain branches, it is possible for plans and subgoals to have already been removed. To prevent this from causing the retraction to fail a disjunction finishing with `true` is included as shown below.

```
strip(P):-
    subtree(P,T),!,
    stripTree(T),
    retract(node(P,_,_,_,_)).

stripTree([]).
stripTree([H|T]):-
    (((retract(node(H,_,_,_,_))); retract(node(H,_))); true),
    stripTree(T).
```

An evaluation of the Constraint Satisfaction Problem (CSP) gives each plan that is considered a number that can be used to sequence the plans. A global finite-domain variable is created for each of the plans to store a value in the domain of plans, ranging from 0 to the number of plans. A solution is a valid sequence where the goals adopted would be achieved if the plans were executed in the order specified by the evaluation. A tree scheduling predicate, `treeScheduler` defined below, is applied to the plan variables to ensure the tree structure is maintained when considering the order in which to execute plans, forming the basis of any scheduling over the plans. This includes preconditions and effects of plans between different branches within a tree to ensure a plan is not scheduled to execute before the plan producing the necessary preconditions has been scheduled to execute.

```
treeScheduler([]).
treeScheduler([[P1,P2]|T]):-
    g_read(P1,I),
    g_read(P2,J),
    I#<#J,
    g_assign(P1,I),
    g_assign(P2,J),
    treeScheduler(T).
```

In many cases, the ordering between subsets of the plans is not important as they will not affect each other in any way, so these plans can safely be given the same sequence number. When executing the plans, this could be seen as either executing them in parallel or executing them in sets, such that all the plans with sequence number 1 are executed before those with sequence number 2, and so forth. By not specifying an exact ordering of the plans, the agent is able to maintain a lot of its autonomy when selecting which plan to execute next. Essentially the “ordering” of plans indicates to the agent which plans are safe to execute together, grouping them into “safe” sets. Provided the agent completes all the plans within one group before moving on to the next, there should be no interference between the various goals. In the worst case, where there was a lot of interference between all of the goals, each plan could be assigned a unique number from their domain of values, specifying the exact ordering in which the plans must be executed for the agent to be successful.

When searching for valid solutions to the goal-plan tree problem, the query is directed from the `reasoning` predicate shown below. When a solution is found, each of the parameters in the head of the predicate is unified with part of the solution or details about the solution for evaluation purposes. This includes counting the number of plans used, the number of goals achieved and the time taken for the solution to be found. The Prolog predicate `real_time(Time)` is used to obtain start and end timings for the evaluation of a goal-plan tree model.

```
reasoning(Schedule, Plans, PlanCount, TimeTaken,
    GoalsSet, GoalsAchieved):-
    real_time(Start),        % start timing the reasoning

    findall(G,root(G),Goals),
    length(Goals, GoalsSet),

    branchOptions,
                                % positive interaction reasoning
    findall([Pa,Pb],pos(Pa,Pb),Merge),
    posScheduler(Merge),
                                % resource reasoning
    branchList(Goals,SumList),
    sort(SumList,SortedSumList),
    resReasoning(SortedSumList),

    findall(P,node(P,_,_,_,_),Plans),
    length(Plans,PlanCount),
    varSetup(Plans,PlanCount),

    findall([Px,Py],tree(Px,Py),A),
    reverse(A,A2),
    treeScheduler(A2),
                                % negative interference reasoning
    findall([Pc,Pd,Pe],neg(Pc,Pd,Pe),Neg),
    negScheduler(Neg),

    varResult(Plans,Schedule),
    fd_labeling(Schedule,[variable_method(standard)]),

    real_time(End),             % reasoning finished
    TimeTaken#=End-Start,
    findall(G2,root(G2),Goals2),
    length(Goals2,GoalsAchieved).
```

The first step in the clause unifies the variable `Goals` with a list of all the top-level goals. The length of this list is queried to identify how many goals have been defined at the start. When reasoning about consumable resources, it is likely that not all goals will be achieved, so a repeat of this is performed to count the number of goals after the actual reasoning and scheduling components of this predicate have been completed. Once the list of goals has been unified, the reduction of the goal-plan trees can start by removing the branch options as described above. The predicates for the three types of reasoning as shown above can be added or removed as necessary, depending on the types of reasoning desired.

Once all the plans that are not required have been removed, either because of branch options, positive interactions or limited resources restricting the number of goals that can be adopted, the finite domain variables for each of the remaining plans are asserted as global variables. This is contained within a `varSetup` predicate that iterates through the list of all the remaining plans asserting the global variables with the domain ranging from 0 to the number of plans now being considered, i.e. the length of the list of plans. After this has been successfully completed, it is then possible to start applying the constraints that restrict the assignment of the values from the domains to the variables. This starts with the scheduling based on the tree structure and finishes with the negative interference reasoning (Section 2.5), when this is incorporated into the types of reasoning being performed. At this point, the labelling of variables with values is to be performed, so the `varResult` predicate simply collects all of the finite domain variables back into a list which is then passed on to the finite domain labelling predicate (`fd_labeling`). This predicate is part of the prolog library for solving finite domain constraint satisfaction problems, and provides a selection of heuristics for ordering the variables; the heuristic selection is given as a parameter to the predicate along with the list of variables.

While this design achieves the objectives of representing and reasoning about the goal-plan tree, it may still be possible to optimise some of the constraints in order to improve their efficiency, thereby reducing the length of time taken for a solution to be found.

2.3 Consumable-Resource Reasoning

The reasoning described here is limited to that of consumable resources rather than reusable resources. The purpose of the reasoning is to restrict the number of goals adopted to those that can be achieved with the amount of consumable resources available and to endeavour to make the best use of those resources through the careful selection of plans when there is a choice between which plans to use in order to achieve the desired result. The reasoning about consumable resources makes use of a small amount of generated summary information to perform this reasoning.

As described in the section above, the resource requirements for each plan are represented by a list of pairs consisting of resource type and quantity required. The total available resources for each type are each defined using a `resource` predicate. This predicate is defined to be dynamic so that when reasoning about resources the quantity available can be updated with the new quantities as they are consumed.

The first part of the resource reasoning is incorporated into the constraint reasoning for the selection between lists of plan options for achieving a goal or subgoal. For each of the plans listed as being an option, a summary of the resource requirements for the sub-tree with the plan at its root is generated. At this point, a single number for all the resource quantities required regardless of resource type is used to decide which plan to use. It is possible to extend the reasoning here to incorporate weightings into the summation of resource requirements in order to indicate preference for the use of certain resources over others.

When this type of reasoning is included, the definition of the `branchStrip` predicate shown above is extended to refer to a predicate that pairs the summary resource requirement with each plan in the list of options. The list of plan options is sorted so that the subgoal branches nearest the leaves at the bottom of the tree are considered first. This is to reduce the number of plans being considered at each iteration through the list and to allow for simpler predicates summing the resource requirements as they do not need to consider branches at lower subgoals. Once the list of plan options paired with resource requirements is formed, it is then sorted into order of increasing resource requirements so the first element in the list is the preferred plan and the remaining plans can again be retracted.

```
branchStrip([]).
branchStrip([H|T]):-
    branchList(H,L),
    sort(L,[_|T2]),
    rmBranch(T2),
    branchStrip(T).

branchList([],T):-T=[].
branchList([P|T],T1):-
    branchList(T,T2),
    subtree(P,X),
    resAll(S,X),
    append([S/P],T2,T1).
```

The `resAll` predicate starts by producing a single long list of the resource requirements for each plan. For each plan, this takes the pairs representing the type of resource and quantity required and appends them to a list of all the resource requirements for the sub-tree being considered. Once all the resource requirements have been compiled into one list, this is sent to a summing predicate to simply add together all the quantities to produce a total resource requirement. It is in this final predicate where weightings could be included, if necessary, to indicate any preferences for which types of resources should be saved or used the most.

```
resAll(S,Ps):-
    resourceList(L,Ps),
    resSum(S,L),!.

resourceList(L,[]):-L=[],!.
resourceList(L,[SG|T]):-      % Only interested in plans
    node(SG,_),
    resourceList(L,T).
resourceList(L,[P|T]):-
    node(P,_,_,_,R),
    resourceList(L1,T),
    append(L1,R,L).

resSum(S,[]):- S=0,!.
resSum(S,[_/X|T]):-
    S#=X+S1,
    resSum(S1,T).
```

After the plan options have been removed, the resource reasoning is next used to consider which goals can be safely adopted given the quantity of each resource available. The reasoning is performed in this order firstly to reduce the number of plans being considered and secondly to allow the summary information generated for reasoning about goal adoption to represent the actual requirements of the goal.

The list of top-level goals can be sorted in the same manner as the list of plan options for selecting the plans or, in this case, goals with the lowest resource requirements. To do this, the first step, as before, is to generate the list of plans in the tree for each goal. This can be performed using the `branchList` predicate with a list of the top-level goals. This will pair up each of the goals with a number representing the sum of resource requirements regardless of type. It is possible to apply different orderings to the list of goals to indicate the importance of a goal, thereby preferring to complete less goals of greater importance than to achieve more goals of less importance. If the order in which the goals are considered for adopting is not important, or if the order is predefined as the order in which the goals were defined, this step can be skipped. This will also provide a decrease in the number of steps and hence the length of time taken to evaluate the problem each time a solution is to be found. In the evaluation of this approach, both sorting and ordering were included in the reasoning.

The main reasoning about resources for goal adoption requires summary information broken down by the different types of resources required. This is so that the reasoning can check that there is actually sufficient resources available for each goal to be adopted. For each goal in the list, the summary information separating the different types of resource information is generated. While the `resAll` predicate produces a combined summary of each of the resource types into one number, the `resType` predicate used here keeps the different types of resources separate when generating the summary information. The summary information produced by the predicate `resType` is an unsorted list containing each of the resource types and the quantity of it required by the goal, for example `S = [r3 / 6, r2 / 5, r1 / 7, r5 / 0, r4 / 0]`. From this list, each of the types of resource is extracted and compared to the available quantity of that resource.

```
resReason(G):-
   goalPlans(G,P),
   resType(S,P),                 % generate resource summary by type
   member(r1/A,S),               % unify the resource values
   member(r2/B,S),
   ...
   resource(r1,RA), RA#>=A,      % check sufficiently available
   resource(r2,RB), RB#>=B,
   ...                           % reserve resources
   retract(resource(r1,RA)),
   NewRA #= RA-A, asserta(resource(r1,NewRA)),
   retract(resource(r2,RB)),
   NewRB #= RB-B, asserta(resource(r2,NewRB)),
   ...
```

If each type of resource has sufficient resources available then the predicate `resReason` will succeed and the quantity of each of the resources available will be lowered accordingly. If one or more types of resource has as insufficient quantity available then the predicate will fail and the *if-then-else* construct from which the predicate was queried (`resReason(G) -> true; strip(G)`) will step to the *else* component where the whole goal will be dropped in the same way as for removing the sub-tree of a plan that is not required. After all the goals have been considered, adopting those that are safe to start, and removing those which are not, the reasoning then returns to the core part of the goal-plan tree representation to schedule the plans for the goals that have been adopted.

2.4 Positive Interaction Reasoning

The positive interaction reasoning attempts to identify plans in different goal-plan trees that can be "merged", as they produce the same effects. When referring to plan merging, it is actually possible to achieve the effects by only using one of the two plans. By doing this, the number of plans required to achieve all the goals adopted can be significantly reduced, especially as the sub-trees of the plans that are not used are also removed when the two plans are merged. If the interaction between the goals occurs at high levels of the goal-plan trees, i.e. near the root with each plan itself having a large sub-tree, then the impact of the merging is particularly significant. Figure 3 illustrates where two plans will achieve the same effect, so only one of the plans is need to reach the desired state.

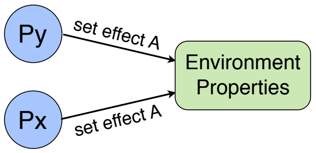


Fig. 3. Illustration of positive interaction

To perform the reasoning in Prolog, a predicate is defined that identifies pairs of plans that produce the same effects by checking that the lists of effects for the two plans are equivalent. This starts by unifying two plans and the list of effects generated by each of the plans, checking that the two plans are not the same plan. The reasoning cycle in Prolog when requested for all pairs of positively interacting plans will iteratively test every pair of plans. For pairs of different plans, the effects of the plans are considered to identify if there is any possibility of merging them. Firstly, it is checked that the list of effects for the first plan is not empty, otherwise all plans that themselves do not achieve effects could be included for merging. Where an effect is produced by Px, the list of effects for the two plans are compared to see if they are equivalent. If so, then with all the constraints satisfied, the pair of plans is returned as a pair of positively interacting plans that can be merged. If the effects are not equivalent, then the solver backtracks to try another pairing until all possible pairings have been tested.

```
pos(Px,Py):-
    node(Px,_,_,XEffects,_),
    not(XEffects=[]),
    node(Py,_,_,YEffects,_),
    Px\=Py,
    seteq(XEffects,YEffects).
```

The findall([Px,Py], pos(Px,Py), Merge) predicate is used to generate a list all the pairs of plans where it is possible for them to be merged. The template used to form the list from the solutions to the pos(Px,Py) predicate creates a sublist for each solution pair of plans. The complete list of positively interacting plans is then used to select and remove plans that are not needed as the effects they produce are duplicated by other plans. By default, the second plan in the pair of interacting plans is retracted, however this is not always the case.

While in the positive interaction reasoning considered here all the effects in the list must match for the plans to be considered for merging, it is also possible to consider a weaker version of positive interaction where only some of the effects match. In this case, in order to ensure that a plan that is kept from the merging with another plan is not then deleted by a later merging, the plan is "marked". This is done by asserting the predicate mark(Plan) for each of the plans that have been kept from a merged pair. When a pair is first considered, it is checked to see if either plan is already marked. If both plans are already marked, then neither plan can be safely removed as it is possible that the intersecting effect that was used to identify the two plans as positively interacting is different to the intersecting effects from the interactions where they have already been "merged".

As the reasoning here checks that the effects are equivalent, it is not necessary to check if one or both plans are already marked. This is because if one plan is marked, and has appeared in more than one positive interaction then the effects of three or more plans must all be equivalent, therefore only one plan is still needed to achieve the effects on behalf of all of the plans. However, as merges could have occurred within the sub-tree of one or both of the interacting plans, it is still necessary to mark the plan kept from a merge to ensure it does not get removed as part of a sub-tree.

The posScheduler predicate defined below starts by checking that the two plans both still exist, i.e. that one or both have not already been removed by other merges. The sub-trees of each plan are then generated to check for any marked plans within the sub-trees that could prevent one of the plans from being removed in a merge. If just one of the plan's sub-trees contains a marked plan, then that plan can be kept while the other is retracted, otherwise neither plan and their sub-trees can be removed.

```
posScheduler([]).
posScheduler([[P1,P2] | T ]):-
   node(P1,_,_,_,_), node(P2,_,_,_,_),
   subtree(P1,X), subtree(P2,Y),
   not((member(XP,X), mark(XP));
       (member(YP,Y), mark(YP))),
   ((not(member(XP,X), mark(XP)), asserta(mark(P1)), strip(P2));
   (not(member(YP,Y), mark(YP)), asserta(mark(P2)), strip(P1))),
   posScheduler(T).
```

When the reasoning about positive interactions is combined with that of reasoning about consumable resources, then the selection for which plan to keep and which plan to drop is influenced by the summary resource requirements for the sub-tree of each plan. In this case the predicate `resAll` is used to produce the summary information for the sub-tree of each of the two plans. The plan with the lower resource requirements is then kept when there is a free choice between the two plans as neither sub-tree contains any marked plans.

The positive interaction reasoning is incorporated into the set of constraints after the branch options have been removed. This is to reduce the number of matches as the branches provide different sets of plans for achieving the same effects within a goal-plan tree.

2.5 Negative Interference Reasoning

While the reasoning about positive interaction identifies plans that produce the same effects, the reasoning about negative interference identifies sets of three plans where one plan generates the effect required by the second plan, and the third plan produces an opposite effect that if it were executed between the first two would cause interference. This can be thought of as a causal link between the first two plans, which the third plan would break. Figure 4 illustrates a case of negative interference.

In Prolog, in order to identify the negative interactions between plans, the `neg(Px,Py,Pz)` predicate is defined to find pairs of plans that have causal links and the plans that can interfere with those links. `Px` is the plan that starts the causal link by producing the desired effect required as a precondition for plan `Py`. Once `Py` has executed, it is assumed that the effect is no longer required, so can

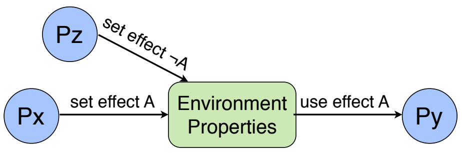


Fig. 4. Illustration of negative interference

be safely altered by other plans such as `Pz`. If however `Pz` attempts to execute between `Px` and `Py`, then this will cause interference, possibly leading to plan and then goal failure. As with the positive interaction reasoning, it is important to check that the plans are all different before comparing the preconditions and effects of the plans. To compare the effects, it is important to split up the pair notation for representing the effects of plans into the two component parts, the factor identifier and the value representing its current state (e.g. `e1/7`). The `member(Element, List)` predicate, in the reasoning predicate shown below, unifies properties of the environment that are common to all three plans but where the value assigned to that property is different in the interfering plan to the value used by the linked plans.

```
neg(Px,Py,Pz):-
    node(Px,_,_,XEffects,_),
    node(Py,_,YPrecon,_,_), Px\=Py,
    node(Pz,_,_,ZEffects,_),
    Px\=Pz,Py\=Pz,
    member(V/N1,YPrecon),
    member(V/N1,XEffects),
    member(V/N2,ZEffects),
    N1#\=N2.
```

This predicate is again queried with the `findall([Px,Py,Pz], neg(Px,Py,Pz),Neg)` predicate to generate a list of all the possible instances of the interference so they can be scheduled to ensure the interference is avoided. For this, the interfering plan either needs to be scheduled to execute before the other plans or after both have executed so the effect is no longer required. This is handled by the `negScheduler` predicate shown below.

```
negScheduler([]).
negScheduler([[Px,Py,Pz]|T]):-
    g_read(Px,A),
    g_read(Py,B),
    g_read(Pz,C),
    A#<#B,(C#<#A;C#>#B),
    g_assign(Px,A),
    g_assign(Py,B),
    g_assign(Pz,C),
    negScheduler(T).
```

The `negScheduler` predicate refers to the finite domain global variables that have been defined for representing the domain of values that can be assigned to each of the variables representing the plans for generating a schedule. The plan producing the effect (`Px`) must always occur before the plan using the effect (`Py`). However, it is possible to schedule the interfering plan (`Pz`) to either execute before `Px` or after `Py`, as long as it does not execute between the two plans.

The reasoning about negative interference is incorporated into the set of constraints after the tree scheduling has been performed. This is to ensure the minimum number of plans are considered as the evaluation of the `neg(Px,Py,Pz)` predicate considers all the possible combinations of three plans. In addition, the main purpose of the negative reasoning is to schedule potentially interfering plans to ensure they do not interfere, rather than reducing the number of plans, so this "scheduling" is performed after all the surplus plans have been removed and the schedule refined based on the constraints in the tree structure.

3 Experimental Results

To compare the performance of the three types of reasoning under different conditions, three different tree structures were used; a deep tree, a broad tree and a tree that is part way between the two (referred to as the general tree structure). The results presented here are a subset of a large set of experiments comparing a wide range of variables covering goal-plan tree size, goal interaction levels and resource availability amongst others. The aim of the experiments was to stress test the approach described here and compare it to the approach described in [5,6], to identify settings where one approach was able to perform better than the other. Each of the types of reasoning was considered independently before combining all three together. The performance of the two reasoning approaches was also compared to the performance without any reasoning, simulated by a Petri net model with the reasoning removed. An example of a more concrete application to which this reasoning could be applied is presented in [5], where a simplified Mars Rover is modelled. While the approach here can be applied to this example, the results presented here are aimed at illustrating performance under highly constrained conditions with a large number of substantially sized goals.

In order to fully evaluate the performance of this approach and compare it to other approaches, a set of large goal-plan trees has been designed with high levels of interactions between them and heavy resource requirements. The goals were designed to test different properties of the reasoning, for example there is a deep tree structure that has very little branch options, and was designed to test the effect of depth and size of sub-trees on the reasoning. Conversely, a broad tree structure containing a lot of branch options has been designed to test the ability of the two approaches to handle branches and select the best options where appropriate. A third tree structure was also used to test the scalability of the two models, so it contained nearly 100 plans and was used in experiments focused on increasing the number of goals.

An overview of the results are presented below, summarising results across the different tree structures. The graphs below combine the results for common settings in each of the tree structures for each type of reasoning, individually and combined. They show the results for experiments using a medium-sized deep and broad tree ($\sim$50 plans) or a large tree ($\sim$100 plans) from the general tree structure, 20 goals, low level resource availability, positive interaction at a high level in the goal-plan tree, negative interference over a long duration and high goal interaction. When showing the timings, the load timings for both models

Fig. 5. Legend for graphs comparing performance over the three different tree structures

are included in the graphs, as well as the run times for the two approaches, as the run time for the Petri net model was very short, but the load time was quite long. The legend for the graphs is shown in Figure 5.

3.1 Reasoning about Consumable Resources

While the Petri net model was able to match the number of goals achieved by the constraint model in the deep tree, the performance in the broad and general trees was much worse, see Figure 6. In comparison, the random Petri net model was able to achieve more goals in the broad and general trees than in the deep tree. The timings for the Petri net model were greater than those for the constraint model when including loading times, especially in the large-sized general tree structure experiments. Overall, the constraint model gave better results both in terms of time and number of goals achieved when there is limited availability of consumable resources, especially in trees where there is a large amount of branching.

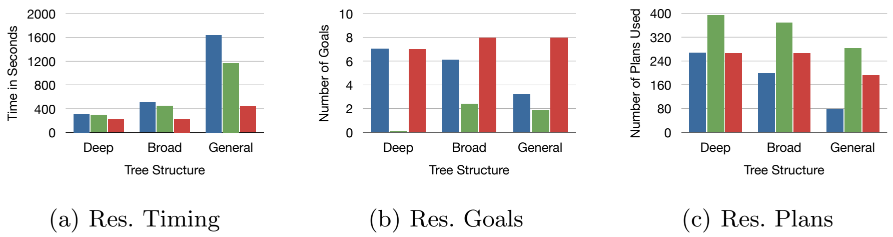
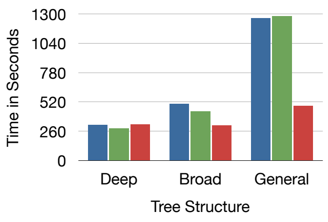


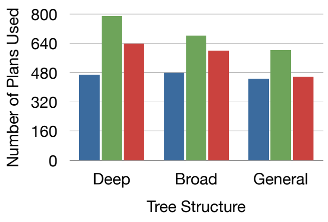
Fig. 6. Comparison results for reasoning about resources across the three tree structures

3.2 Reasoning about Positive Interaction

When reasoning about positive interaction, the Petri net was able to generate better results based on the reduction in the number of plans used in each of the tree structures, see Figure 7. Comparing the timings here shows that while the time taken between the Petri net and the constraint models was the same for the deep tree, the Petri net model took longer to load in the experiments for the other two tree structures, especially the large tree size of the general tree structure. When the number of plans used is the key criteria, then the Petri net



(a) Pos. Timing



(b) Pos. Plans

Fig. 7. Comparison results for reasoning about positive interaction across the three tree structures

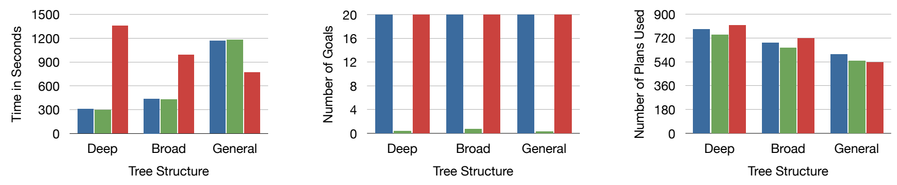
model performs better; however, if time is critical then the constraint model can produce results slightly faster when reasoning about positive interaction is applied.

3.3 Reasoning about Negative Interference

While the reasoning about negative interference was the most time consuming of all the three types of reasoning, it is perhaps the most critical when comparing the results achieved to those produced when no reasoning is included, as illustrated in Figure 8. In this case, the time taken by the Petri net even when the load times are included is much shorter for the experiments on the deep and broad tree structures. However, the loading time on the large-sized tree for the general tree structure does take longer than the constraint model in this setting. Overall, the Petri net model offers better results here, especially with the small and medium tree structures.

3.4 Combined Reasoning

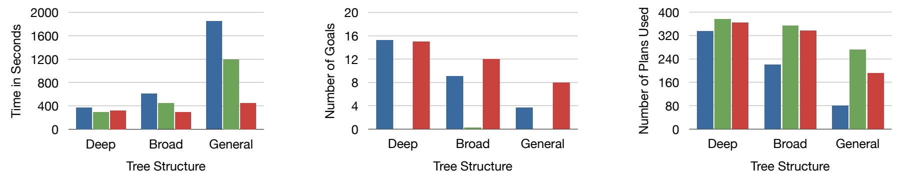
When combining the three types of reasoning together, the number of goals achieved increased, especially in the deep tree where a large number of plans were saved by the positive interaction reasoning, as shown in Figure 9. The resources that would have been consumed by these plans were then available for use in achieving other goals. This combined effect is less noticeable in the broad and general trees. However, the constraint model was generally able to make the most optimisations here. The exception to this is that as the availability of the resources was increased in the general tree structure, the number of goals started and hence the plans interacting increased, resulting in more plans not being used so more resources being saved for use in achieving further goals. In the high level resource availability for the general tree, this lead to all goals being achieved by the Petri net model.



(a) Neg. Timing (b) Neg. Goals (c) Neg. Plans

Fig. 8. Comparison results for reasoning about negative interference across the three tree structures

In the experiments for the deep tree, the Petri net timings even when including the loading times were quite similar to those for the constraint model, however in the experiments for the other two tree structures, especially the large-sized general tree, the time taken for loading the Petri net model was greater than the time taken for the constraint model to find a solution. Despite the additional time taken for the reasoning in both models, the benefits gained from performing the reasoning over those shown in the random Petri net model indicate that it is worth considering taking the time to find a good solution. In highly dynamic environments, there may not be the time available to consider this as too much would have changed by the time a simulation had finished.



(a) Combi. Timing (b) Combi. Goals (c) Combi. Plans

Fig. 9. Comparison results for combined reasoning across the three tree structures

When increasing the number of goals used in the general tree structure from 20 to 50, the Petri net model became too large for the Petri net editor (Renew [4]) to load the model, while the constraint-model was able to continue to reason about up to 75 goals. However, it took 9hrs to find a solution, so further increases in the number of goals were not tested. It should be noted that in current agent applications, the size and number of goals tends to be significantly less than those tested here. This means that the time taken for the reasoning can be within acceptable ranges for practical purposes, at least in some applications.

4 Conclusions and Future Work

In this paper we have presented a specific Prolog implementation (with constraint solving) that could be used to solve the goal-plan tree problem, which has then been compared to a second specific implementation based on Petri nets. While these two approaches are both capable of solving the same problems, the techniques they use are different and as a result the solutions they offer can vary. For example, the approach here controls the sequence in which goals and plans are considered. If a set of goals is evaluated again, it will be evaluated in the same order and give the same answer each time. However, in the Petri net approach, no exact order in which goals and plans are evaluated is set, so in each evaluation the order can vary. This can lead to differences in the results returned, particularly when resources are constrained. It is possible that variations in the used of the underlying techniques would offer different advantages, however further experimental comparisons such as the one performed here would be needed.

The approach presented here has been experimentally compared to the Petri net approach described in [6]. The complete results from the comparison can be found in [7]. The aim of these experiments was to test the two approaches under highly constrained conditions and to identify situations where one approach may be better suited over the other. The differences between these two approaches can be beneficial in different situations and conditions where some properties of one or the other approach may be preferable.

The results presented here show that while the Petri net model has faster running times, it also has the slowest loading times with the greatest memory usage once loaded. One of the side effects of this is that, as the size of the goal-plan trees or the number of top-level goals increases, the load times rapidly increase until the application running the Petri net simulations is no longer able to load the Petri net goal-plan tree representation. Refinements and changes in the way the goals are represented may reduce the problem allowing greater numbers of goals to be handled in the Petri net model. Similarly, it is possible that refinements in the efficiency of the Prolog constraints used in the constraint-based model may improve the performance of this model as well.

In some cases the Petri net model can give better results over the constraint-based model. In particular, when reasoning about positive interactions between the goals, the Petri net model gives better results in terms of the number of plans used, and when reasoning about negative interference, the Petri net model also gives faster results for successfully achieving all goals, even when including the loading times. The structure of the tree also affected the performance of the two reasoning models, with the constraint-based model performing better when applied to reasoning about the limited availability of resources in trees where there is a large amount of branching and little depth.

Where the ability to reason about large numbers of goals is required, especially for large sized trees, the constraint model demonstrated that it was able to scale and find solutions to larger problems. However, the trade-off comes at the time taken, for example taking 9 hours to reason about 75 goals in one setting.

The approach described here has been compared based on the individual types of reasoning and the combined reasoning. While in most cases it makes sense to combine all three types of reasoning, there may be application areas where only one is needed. For example, in applications where there is limited availability of consumable resources but very little interaction between the goals it may only make sense to use the resource reasoning. Similarly, in applications where there are a lot of common goals to achieve the same effects, and abundant resources it may be better to use only the positive interaction reasoning. In applications where there is likely to be a lot of conflict between the goals or where it is more critical that all the goals are achieved, but again with abundant resources, it may be sufficient to just apply the negative interference reasoning.

In conclusion, the following recommendations can be made to agents about which model they may wish to consider depending on their specific circumstances:

– When just considering resource reasoning, if the goal-plan trees contain a lot of branching then the constraint-based model gives better results in terms of goals achieved.
– When just considering positive interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the reduction in plans used.
– When just considering negative interaction reasoning, the Petri net model gives better results for all goal-plan tree structures in terms of the time taken to perform the reasoning.
– When considering the combination of all three types of reasoning, the constraint-based model gives better results in terms of goals achieved except when there is high resource availability, in which case the Petri net model performs better.
– When there are a large number of large goals (i.e. 50 or more goals containing more than 100 plans), only the constraint-based approach is able to perform the reasoning, although it will take considerable time to find a solution.

The reasoning about resources that has been considered here has focused on consumable resources that are limited in their availability. Another type of resource that is often used are reusable resources, such as communication channels. A model was shown in [6] of how this could be incorporated into the Petri net model, and constraints could be added into the constraint-based approach to prevent two plans attempting to use the same reusable resource at the same time. This was not initially included as the use of these resources can be easily scheduled, while the use of consumable resources has greater restrictions applied to it. The reasoning about consumable resources is also the more difficult of the two types of resources to implement, being possible to later incorporate the reasoning for reusable resources easily. In addition, when considering consumable resources, all the goals are currently assumed to consume resources without any goals to recharge them or to create more resource instances. The Petri net approach and to some extent the constraint-based approach are however robust enough to handle this, at least in a simplistic manner. However, further work to extend both approaches to allow for more generic maintenance goals rather than only achievement goals is required.

References

1. Clement, B.J., Durfee, E.H.: Identifying and resolving conflicts among agents with hierarchical plans. In: Proceedings of AAAI Workshop on Negotiation: Settling Conflicts and Identifying Opportunities, Technical Report WS-99-12, pp. 6–11. AAAI Press, Menlo Park (1999)
2. Clement, B.J., Durfee, E.H.: Theory for coordinating concurrent hierarchical planning agents using summary information. In: AAAI 1999/IAAI 1999: Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence, pp. 495–502. American Association for Artificial Intelligence, Menlo Park (1999)
3. Clement, B.J., Durfee, E.H.: Performance of coordinating concurrent hierarchical planning agents using summary information. In: Proceedings of 4th International Conference on Multi-Agent Systems (ICMAS), pp. 373–374. IEEE Computer Society, Boston (2000)
4. Kummer, O., Wienberg, F., Duvigneau, M.: Renew – the Reference Net Workshop, Release 2.1 (May 2006)
5. Shaw, P.H., Bordini, R.H.: Towards alternative approaches to reasoning about goals. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2007. LNCS (LNAI), vol. 4897, pp. 104–121. Springer, Heidelberg (2008)
6. Shaw, P., Farwer, B., Bordini, R.H.: Theoretical and experimental results on the goal-plan tree problem. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, vol. 3, pp. 1379–1382 (2008)
7. Shaw, P.H.: Reasoning about Goal-Plan Trees in Autonomous Agents: Development of Petri net and Constraint-Based Approaches with Resulting Performance Comparisons. PhD thesis, School of Engineering and Computing Sciences, University of Durham, UK (January 2010)
8. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and avoiding interference between goals in intelligent agents. In: Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI), pp. 721–726. Morgan Kaufmann, Acapulco (2003)
9. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and exploiting positive goal interaction in intelligent agents. In: Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 401–408. ACM Press, New York (2003)
10. Thangarajah, J., Winikoff, M., Padgham, L.: Avoiding resource conflicts in intelligent agents. In: Proceedings of 15th European Conference on Artifical Intelligence (ECAI 2002), IOS Press, Amsterdam (2002)

Intention Change via Local Assignments[⋆]

Hans van Ditmarsch[1,⋆⋆], Tiago de Lima[2], and Emiliano Lorini[3]

[1] University of Sevilla, Spain
[2] CRIL, University of Artois and CNRS, France
[3] IRIT, CNRS, France

Abstract. We present a logical approach to intention change. Inspired by Bratman's theory, we define intention as the choice to perform a given action at a certain time point in the future. This notion is modeled in a modal logic containing a temporal modality and modal operators for belief and choice. Intention change is then modeled by a specific kind of dynamic operator, that we call 'local assignment'. This is an operation on the model that changes the truth value of atomic formulae at specific time points. Two particular kinds of intention change are considered in some detail: intention generation and intention reconsideration.

1 Introduction

According to Bratman's planning theory of intention [6], rational agents build complex plans and organize their life on the basis of sequences of actions. They intend to perform certain actions and plans because they have reasons to perform them, and they write these actions and plans on their mental agenda, in order to remember when to perform them. In other terms, rational agents settle themselves in advance on plans for the future. That is, they have *future-directed intentions*. But also, they intend to do things here and now. That is, they have *present-directed intentions*. In such situations, they initiate the intended action and sustain it until its completion. As the time goes on, rational agents keep their future-oriented intentions unless they have no more reason to perform in the future what they intend to do. Hence, they reconsider their plans and possibly change them.

So, a future-directed intention has its own life in the mind of an agent. There is an initial moment in which it is generated. As time goes on, it may be reconsidered and eventually dropped. But, it may also last until it transforms into a present-directed intention, which is responsible for initiating the agent's action.

Since the seminal work of Cohen & Levesque [9] aimed at implementing Bratman's theory of intention, many formal logics for reasoning about intentions and plans, and for describing their dynamics have been developed (see, *e.g.* [23,27,22,29,20,2,7,10,18,15]). Most of them are based on dynamic logic extended by doxastic modal operators, and by modal operators for motivational attitudes, such as preferences, goals and intentions. These logics are traditionally called BDI (belief, desire, intention) logics. But, although

[⋆] This paper is an extended version of [21].

[⋆⋆] Hans van Ditmarsch is also affiliated to the Institute of Mathematical Sciences Chennai (IMSC), as associated researcher.

logical analysis of intention and plan dynamics are available in the literature, the issue of a *formal semantics* for the dynamics of intentions and plans has received much less attention. Indeed, all previous approaches are mostly interested in characterizing in the object language the epistemic conditions under which an agent's intention persists over time, and the epistemic conditions under which an agent's intention is generated. However, they do not provide a semantic characterization of the process of generating an intention and of the process of reconsidering an intention.

The aim of this work is to shed light on this unexplored area by proposing a formal semantics of intention and plan dynamics based on the notion of *local assignment*. The function of a local assignment is to change the truth value of a given proposition at a specific time point along a history. We combine a static modal logic including a temporal modality and modal operators for mental attitudes belief and choice with three kinds of dynamic modalities and corresponding three kinds of local assignments: local assignments operating on an agent's beliefs, local assignments operating on the agent's choices and local assignments operating on the physical world. An agent's intention is defined in our approach as the agent's choice to perform a given action at a certain time point in the future, and two operations on intentions called *intention generation* and *intention reconsideration* are defined as specific kinds of local assignments on choices.

The rest of the paper is organized as follows. The first part (Section 2) introduces a static logic of time, action, belief, choice and intention. In the second part (Section 3), we move from a static perspective on agents' attitudes to a dynamic perspective, by adding the notion of local assignment to the logic of Section 2. We first present the syntax and semantics of three kinds of assignments: on beliefs, on choices and on the physical world. Then, in Section 4, we focus on two specific kinds of local assignment which allow to model the processes of plan generation and plan reconsideration. In Section 5, we apply our logical framework to a concrete example. Finally, in Section 6, we discuss some related work.

2 A logic of Time, Action and Mental Attitudes

We introduce a modal logic called **L** which supports reasoning about time, action and three different kinds of mental attitudes: beliefs, choices (or chosen goals), and intentions.

2.1 Syntax

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of nonnegative integers. Let $ATM^{Fact} = \{f_1, f_2, \ldots\}$ be a nonempty finite set of atoms denoting facts (or state of affairs). And let $ATM^{Act} = \{\alpha, \beta, \ldots\}$ be a nonempty finite set of atoms denoting actions. The atom α stands for 'the agent performs a certain action α'. We define $ATM = ATM^{Fact} \cup ATM^{Act}$ to be the set of atomic formulae. We denote $p, q, \ldots$ the elements in ATM.

The language $\mathcal{L}$ of the logic **L** is the set of formulae defined by the following BNF:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid [\mathtt{B}]\varphi \mid [\mathtt{C}]\varphi \mid \bigcirc\varphi$$

where p ranges over ATM. The other Boolean constructions $\bot$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined from $\top$, $\neg$ and $\vee$ in the standard way.

The three modal operators of our logic have the following reading: $[\mathtt{B}]\varphi$ means 'the agent believes that φ', $[\mathtt{C}]\varphi$ means 'the agent has chosen φ' (or 'the agent wants φ to be true'), and $\bigcirc\varphi$ means 'φ will be true in the next state, if no event affecting the world occurs'. The operator $\bigcirc$ describes the passive (or inertial) evolution of the world. That is, how the world evolves over time when no event affecting it occurs (this point will be better clarified in Section 3.4). [1] Operator $[\mathtt{C}]$ is used to denote the agent's choices. That is, the state of affairs that the agent has decided to pursue. Similar operators have been studied in [9,20,23]. We write $\bigcirc^n\varphi$ to indicate that the sentence φ is subject to n iterations of the modality $\bigcirc$, where $n \in \mathbb{N}$. More formally, $\bigcirc^0\varphi \overset{\text{def}}{=} \varphi$, and $\bigcirc^{n+1}\varphi \overset{\text{def}}{=} \bigcirc\bigcirc^n\varphi$. The following abbreviation defines the concept of intention for every $\alpha \in ATM^{Act}$ and $n \in \mathbb{N}$:

$$\mathtt{I}^n(\alpha) \overset{\text{def}}{=} [\mathtt{C}]\bigcirc^n\alpha$$

$\mathtt{I}^n(\alpha)$ stands for 'the agent intends to do action α in n steps from now'. Note that, if the agent has the intention to do α only once n steps from now, then after n steps $\mathtt{I}^n(\alpha)$ does not hold anymore. That is, the intention is dropped once n steps have passed.

2.2 Semantics

Definition 1 (L-model). *Models of the logic **L** (**L**-models) are tuples $M = \langle H, \mathscr{B}, \mathscr{C}, \mathscr{V}\rangle$, where:*

- *$H = \{h, h', \ldots\}$ is a nonempty set of possible histories;*
- *$\mathscr{B}$ and $\mathscr{C}$ are two total functions with signature $H \longrightarrow 2^H$ such that for every $h \in H$:*
  (C1) if $h' \in \mathscr{B}(h)$ then $\mathscr{B}(h') = \mathscr{B}(h)$,
  (C2) if $h' \in \mathscr{B}(h)$ then $\mathscr{C}(h') = \mathscr{C}(h)$;
- *$\mathscr{V}$ is a valuation function with signature $ATM \longrightarrow 2^{H\times\mathbb{N}}$.*

For every history h, $\mathscr{B}(h)$ is the set of histories that are compatible with the agent's beliefs at history h (or belief accessible histories at h), and $\mathscr{C}(h)$ is the set of histories that are compatible with the agent's choices at history h (or choice accessible histories at h). Constraint C1 (resp. C2) expresses that the agent's beliefs (resp. choices) are positively and negatively introspective.

Relations $\mathscr{C}$ and $\mathscr{B}$ are defined on histories instead of on points on histories. If the latter approach had been taken, it would be possible that the agent chooses, resp. believes, to be at a different time point than the actual one. To avoid this, it would be necessary to add restrictions to the model in order to "synchronize" choices and believes. The resulting class of models would validate the same formulae as the one we use here.

We call 'pointed model' a pair $M, h(n)$, where M is a model as defined above, $h \in H$ and $n \in \mathbb{N}$.

[1] The logic presented up till here could easily be extended with other temporal operators, such as "until". But, this would make the axiomatization of its extension, presented in Section 3.3, more complicated. We prefer to leave it for future work.

Definition 2 (Truth of L-formulae). *The satisfaction relation $\models$, between formulae in* **L** *and pointed models, is defined recursively as follows:*

$$\begin{array}{lll} M, h(n) \models \top & & \\ M, h(n) \models p & \textit{iff} & (h, n) \in \mathscr{V}(p) \\ M, h(n) \models \neg\varphi & \textit{iff} & \textit{not } M, h(n) \models \varphi \\ M, h(n) \models \varphi \vee \psi & \textit{iff} & M, h(n) \models \varphi \textit{ or } M, h(n) \models \psi \\ M, h(n) \models \bigcirc\varphi & \textit{iff} & M, h(n+1) \models \varphi \\ M, h(n) \models [\mathtt{C}]\varphi & \textit{iff} & M, h'(n) \models \varphi \textit{ for all } h' \in \mathscr{C}(h) \\ M, h(n) \models [\mathtt{B}]\varphi & \textit{iff} & M, h'(n) \models \varphi \textit{ for all } h' \in \mathscr{B}(h) \end{array}$$

We write $\models_{\mathbf{L}} \varphi$ to denote that φ is *valid* (*i.e.* φ is true in all **L**-pointed models). We say that φ is *satisfiable* if and only if $\neg\varphi$ is not valid.

2.3 Axiomatization

Fig. 1 (on page 140) contains the axiomatization of the logic **L**. We have Axioms K, D, 4 and 5 for beliefs and Axioms K and D for choices (as in [9]). Thus, we assume that if an agent believes (resp. does not believe) that φ then he believes this (Axioms 4 and 5 for $[\mathtt{B}]$), and we also assume that an agent cannot have inconsistent beliefs (Axiom D for $[\mathtt{B}]$). We assume that an agent cannot have inconsistent choices (Axiom D for $[\mathtt{C}]$). And we also assume that if an agent wants (resp. does not want) φ to be true then he believes this (Axioms $\mathbf{PIntr}_{[\mathtt{C}]}$ and $\mathbf{NIntr}_{[\mathtt{C}]}$). Similar principles of positive and negative introspection for choices are given in [13]. At the current stage, these are the only interaction principles between beliefs and choices. We postpone to future work a refinement of the logic **L** by interaction principles like $[\mathtt{B}]\varphi \rightarrow [\mathtt{C}]\varphi$ (if the agent believes that φ then he wants φ to be true) or $[\mathtt{C}]\varphi \rightarrow [\mathtt{B}]\varphi$ (if the agent wants φ to be true then he believes that φ). Similar principles have been studied for instance in [9,23].

We also have a basic principle for the temporal *next* operator (Axiom $\mathbf{Funct}_{\bigcirc}$): φ will be true in the next state if and only if it is not the case that φ will be false in the next state.

Finally, we have interaction principles between time and beliefs, and between time and goals. These are called *perfect recall* (Axioms $\mathbf{PR}_{[\mathtt{B}]}$ and $\mathbf{PR}_{[\mathtt{C}]}$) and *no learning* (Axioms $\mathbf{NL}_{[\mathtt{B}]}$ and $\mathbf{NL}_{[\mathtt{C}]}$) [14]. According to these four axioms, if the agent believes (resp. wants) that φ will be true in the next state if no event affecting the world occurs then, if no event affecting the world occurs, in the next state the agent will believe (resp. want) that φ and vice-versa.

We call **L** the logic axiomatized by the principles in Fig. 1, and we write $\vdash_{\mathbf{L}} \varphi$ if φ is a **L**-theorem. For example, the following is provable using Axiom 4, Necessitation rule for $[\mathtt{B}]$ and Axiom $\mathbf{PR}_{[\mathtt{B}]}$:

$$\vdash_{\mathbf{L}} [\mathtt{B}]\bigcirc\varphi \leftrightarrow [\mathtt{B}]\bigcirc[\mathtt{B}]\varphi$$

Theorem 1. *The logic* **L** *is completely axiomatized by the principles in Fig. 1.*

(**PC**)	All principles of classical propositional calculus
(**KD45**$_{\mathtt{[B]}}$)	All principles of modal logic KD45 for $\mathtt{[B]}$
(**KD**$_{\mathtt{[C]}}$)	All principles of modal logic KD for $\mathtt{[C]}$
(**K**$_{\bigcirc}$)	All principles of modal logic K for $\bigcirc$
(**PIntr**$_{\mathtt{[C]}}$)	$\mathtt{[C]}\varphi \to \mathtt{[B][C]}\varphi$
(**NIntr**$_{\mathtt{[C]}}$)	$\neg\mathtt{[C]}\varphi \to \mathtt{[B]}\neg\mathtt{[C]}\varphi$
(**Funct**$_{\bigcirc}$)	$\bigcirc\varphi \leftrightarrow \neg\bigcirc\neg\varphi$
(**PR**$_{\mathtt{[B]}}$)	$\mathtt{[B]}\bigcirc\varphi \to \bigcirc\mathtt{[B]}\varphi$
(**PR**$_{\mathtt{[C]}}$)	$\mathtt{[C]}\bigcirc\varphi \to \bigcirc\mathtt{[C]}\varphi$
(**NL**$_{\mathtt{[B]}}$)	$\bigcirc\mathtt{[B]}\varphi \to \mathtt{[B]}\bigcirc\varphi$
(**NL**$_{\mathtt{[C]}}$)	$\bigcirc\mathtt{[C]}\varphi \to \mathtt{[C]}\bigcirc\varphi$

Fig. 1. Axiomatization of **L**

Proof. First, we provide an alternative semantics for **L**, in terms of standard Kripke frames. An alternative model is a tuple of the form $M' = \langle W, R_{\bigcirc}, R_{\mathtt{[B]}}, R_{\mathtt{[C]}}, V\rangle$, where W is a non-empty set of possible worlds, $R_{\bigcirc}$ is a serial and deterministic accessibility relation over W, $R_{\mathtt{[B]}}$ is a serial, transitive and Euclidean accessibility relation over W, $R_{\mathtt{[C]}}$ is a serial accessibility relation over W, V is a valuation function with signature $ATM \to 2^W$, and where the following interaction constraints are satisfied:

(**PIntr**$_{\mathtt{[B]}}$)	if $wR_{\mathtt{[B]}}w'$ and $w'R_{\mathtt{[C]}}w''$ then $wR_{\mathtt{[C]}}w''$;
(**NIntr**$_{\mathtt{[C]}}$)	if $wR_{\mathtt{[B]}}w'$ and $wR_{\mathtt{[C]}}w''$ then $w'R_{\mathtt{[C]}}w''$;
(**PR**$_{\mathtt{[B]}}$)	if $w(R_{\bigcirc} \circ R_{\mathtt{[B]}})w'$ then $w(R_{\mathtt{[B]}} \circ R_{\bigcirc})w'$;
(**PR**$_{\mathtt{[C]}}$)	if $w(R_{\bigcirc} \circ R_{\mathtt{[C]}})w'$ then $w(R_{\mathtt{[C]}} \circ R_{\bigcirc})w'$;
(**NL**$_{\mathtt{[B]}}$)	if $w(R_{\mathtt{[B]}} \circ R_{\bigcirc})w'$ then $w(R_{\bigcirc} \circ R_{\mathtt{[B]}})w'$;
(**NL**$_{\mathtt{[C]}}$)	if $w(R_{\mathtt{[C]}} \circ R_{\bigcirc})w'$ then $w(R_{\bigcirc} \circ R_{\mathtt{[C]}})w'$.

Alternative pointed models are tuples of the form $\langle M', w\rangle$ where M' is as defined above and $w \in W$. The alternative satisfaction relation $\models'$, as well as validity, are defined as usual.

Second, it is easy to see that the axiomatic system in Fig. 1 is sound and complete with respect to the class of alternative models, via the Sahlqvist theorem, cf. [5, Th. 2.42]. Indeed all axioms in Fig. 1 are in the so-called Sahlqvist class [25]. Thus, they are all expressible as first-order conditions on Kripke models and are complete with respect to the defined model classes.

Third, we show that for every alternative pointed model $\langle M', w\rangle$, there is a pointed model $\langle M, h\rangle$, where $M = \langle H, \mathscr{B}, \mathscr{C}, \mathscr{V}\rangle$, such that for every formula $\varphi \in \mathcal{L}$, $M', w \models' \varphi$ if and only if $M, h(0) \models \varphi$. The construction of $\langle M, h\rangle$ is performed in three steps:

Step 1: Unravel the alternative pointed model $\langle M', w\rangle$, in such a way that it becomes an infinite tree with root w and which is bisimilar [5] to the original model M'. Call this new tree-shaped model M''.

Step 2: Label each world v in M'' with a natural number $L(v)$, as follows: (a) $L(w) = 0$. (b) $L(v') = L(v)$, if $v' \in (R_{[B]} \cup R_{[C]})(v)$. (c) $L(v') = L(v) + 1$, if $v' \in R_{\bigcirc}(v)$. Remark 1. Because M' satisfy constraints $\mathbf{NL}_{[B]}$ $\mathbf{PR}_{[C]}$ (resp. $\mathbf{NL}_{[B]}$ and $\mathbf{PR}_{[C]}$), model M'' constructed in steps 1 and 2 has the following property: Let $(v, v'), (u, u') \in R^{+}_{\bigcirc}$ (the transitive closure of $R_{\bigcirc}$) such that $L(v) = L(u)$ and $L(v') = L(u')$. Then, $(v, u) \in R_{[B]}$ (resp. $R_{[C]}$) if and only if $(v', u') \in R_{[B]}$ (resp. $R_{[C]}$).

Step 3: Construct the model $M = \langle H, \mathscr{B}, \mathscr{C}, \mathscr{V} \rangle$ from model M'', as follows: (a) H is the set of branches $h = (v_0, v_1, v_2, \dots)$ of the tree such that $(v_i, v_{i+1}) \in R_{\bigcirc}$, for all $i \geq 0$. (b) $\mathscr{B}$ (resp. $\mathscr{C}$) is the set of pairs of branches (h, h') of H such that there is an arrow labeled by $R_{[B]}$ (resp. $R_{[C]}$) from a world in h to a world in h', and (c) $\mathscr{V}(p)$ is the set of pairs (h, n) such that $h \in H$, v is in the branch h, $L(v) = n$ and $v \in V(p)$.

Remark 2. It follows from the construction of M'' that, if $(h, h') \in \mathscr{B}$ (resp. $\mathscr{C}$) then there is a pair of worlds (v, v') such that $L(v) = L(v')$ and $(v, v') \in R_{[B]}$ (resp. $R_{[C]}$).

Fourth, we show that $\langle M'', w \rangle$ and $\langle M, h(0) \rangle$ are, in some sense, bisimilar.

Forth condition : It is easy to see that, by construction of M, if $(v, v') \in R_{\bigcirc}$ then $L(v') = L(v) + 1$. Moreover, let v be in branch $h \in H$, and v' in branch $h' \in H$. Then, if $(v, v') \in R_{[B]}$ (resp. $R_{[C]}$) then, again by construction, $(h, h') \in \mathscr{B}$ (resp. $\mathscr{C}$).

Back condition : Analogously, it is easy to see that if $L(v') = L(v) + 1$ then $(v, v') \in R_{\bigcirc}$. Moreover, if $(h, h') \in \mathscr{B}$ (resp. $\mathscr{C}$) then, by Remark 1, there is a pair of worlds (v, v') in M'' such that $L(v) = L(v')$ and $(v, v') \in R_{[B]}$ (resp. $R_{[C]}$). Moreover, by Remark 2, for every pair of worlds (u, u') such that u is in branch $h \in H$, u' is in branch $h' \in H$ and $L(u) = L(u')$, we have that $(u, u') \in R_{[B]}$ (resp. $R_{[C]}$).

Fifth, with an induction on the structure of φ we show that $\langle M'', v \rangle \models' \varphi$ if and only if $\langle M, h(L(v)) \rangle \models \varphi$, where v is in branch $h \in H$.

The induction base has two cases. Case 1 is $\varphi = \top$, and Case 2 is $\varphi = p$ for some atomic formula p. Both are straightforward. There are five cases in the induction step, one for each operator of the logic. The cases for the boolean operators $\neg$ and $\vee$ are straightforward.

Case 3 is $\varphi = \bigcirc\varphi_1$. $\langle M'', v \rangle \models' \bigcirc\varphi_1$ iff $\langle M'', v' \rangle \models' \varphi_1$, for all $v' \in R_{\bigcirc}(v)$ iff $\langle M, h(L(v') + 1) \rangle \models \varphi_1$ (by the forth and back conditions above and the induction hypothesis) iff $\langle M, h(L(v)) \rangle \models \bigcirc\varphi$.

Case 4 is $\varphi = [\mathrm{B}]\varphi_1$. $\langle M'', v \rangle \models' [\mathrm{B}]\varphi_1$ iff $\langle M'', v' \rangle \models' \varphi_1$, for all $v' \in R_{[B]}(v)$ iff $\langle M, h'(L(v)) \rangle \models \varphi_1$, for all $h' \in R_{[B]}(h)$, and where v' is in the sequence h'. (again, by the forth and back constructions above and the induction hypothesis) iff $\langle M, h(L(v)) \rangle \models [\mathrm{B}]\varphi_1$.

Case 5 is analogous to case Case 4.

Sixth, the converse can be done as well. That is, for every pointed model $\langle M, h \rangle$ it is possible to construct an alternative pointed model $\langle M', w \rangle$ that satisfies the same formulae of $\mathcal{L}$. This is straightforward, and left to the reader.

$\square$

3 Local Assignments

In this section, we extend the logic **L** of Section 2 by modal operators for physical world change and mental attitude change. We distinguish two kinds of mental attitude change: belief change and choice change. We call $\mathbf{L}^+$ the extended logic. Logic $\mathbf{L}^+$ is based on the notion of *local assignment*. The function of a local assignment is to associate the truth value of a certain formula φ to a propositional atom p at a specific time point n along a history.

3.1 Syntax

We write ASG to denote the set of all partial functions σ with signature $(ATM \times \mathbb{N}) \to \mathcal{L}$. The elements in ASG are called *local assignments*, or simply *assignments*. We write $CASG$ to denote the set of all triples $\Sigma = (\sigma_B, \sigma_C, \sigma_W)$ such that $\sigma_W, \sigma_B, \sigma_C \in ASG$. The elements in the set $CASG$ are called *complex local assignments*, or simply *complex assignments*. Every complex assignment $\Sigma = (\sigma_B, \sigma_C, \sigma_W)$ is composed by a *belief assignment* σ_B (an assignment responsible for belief change), a *choice assignment* σ_C (an assignment responsible for choice change), and a *world assignment* σ_W (an assignment responsible for world change). When spelling out the elements of $\sigma_B = \{(p_1, n_1, \varphi_1), \ldots, (p_m, n_m, \varphi_m)\}$, we write it as $\{(p_1,n_1) \overset{B}{\mapsto} \varphi_1, \ldots, (p_m,n_m) \overset{B}{\mapsto} \varphi_m\}$, and analogously for σ_C and σ_W.

Definition 3 ($\Uparrow\Sigma$). *Let $\Sigma = (\sigma_B,\sigma_C,\sigma_W)$. We define $\Uparrow\Sigma$ as the triple $(\Uparrow\sigma_B, \Uparrow\sigma_C, \Uparrow\sigma_W)$ such that, for all $p \in ATM$ and $n \in \mathbb{N}$:*

- $\Uparrow\sigma_B(p, n) = \sigma_B(p, n + 1)$,
- $\Uparrow\sigma_C(p, n) = \sigma_C(p, n + 1)$, *and*
- $\Uparrow\sigma_W(p, n) = \sigma_W(p, n + 1)$.

This means that the belief/choice/world assignments $\Uparrow\sigma_B/\Uparrow\sigma_C/\Uparrow\sigma_W$ are obtained by, respectively, shifting one step forward the belief/choice/world assignments $\sigma_B/\sigma_C/\sigma_W$.

The language $\mathcal{L}^+$ of the logic $\mathbf{L}^+$ is defined by the BNF:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid [\mathsf{B}]\varphi \mid [\mathsf{C}]\varphi \mid \bigcirc\varphi \mid [\Sigma{:}\textsc{w}]\varphi \mid [\Sigma{:}\textsc{b}]\varphi \mid [\Sigma{:}\textsc{c}]\varphi$$

where p ranges over ATM and Σ ranges over $CASG$.

The formula $[\Sigma{:}\textsc{w}]\varphi$ stands for: 'φ holds in the physical world after the occurrence of the event Σ'. The formula $[\Sigma{:}\textsc{b}]\varphi$ stands for: 'φ holds in the context of the agent's beliefs after the occurrence of the event Σ'. (This does *not* mean that the agent believes φ after Σ. See Theorem 3 for precision.) The formula $[\Sigma{:}\textsc{c}]\varphi$ stands for: 'φ holds in the context of the agent's choices after the occurrence of the event Σ'.

3.2 Semantics

For every **L**-model M, every $n \in \mathbb{N}$ and every $\Sigma = (\sigma_B, \sigma_C, \sigma_W)$, we define the model M_n^Σ which results from the update of M at the time point n by the complex assignment Σ.

Definition 4 (Updated model M_n^Σ). *For every* $\mathbf{L}$*-model* $M = \langle H, \mathscr{B}, \mathscr{C}, \mathscr{V} \rangle$ *and every* $n \in \mathbb{N}$, M_n^Σ *is the tuple* $\langle H_n^\Sigma, \mathscr{B}_n^\Sigma, \mathscr{C}_n^\Sigma, \mathscr{V}_n^\Sigma \rangle$*, where:*

$$
\begin{aligned}
H_n^\Sigma =& \{h_W | h \in H\} \cup \{h_B | h \in H\} \cup \{h_C | h \in H\};\\
\mathscr{B}_n^\Sigma(h_W) =& \{h'_B | h' \in \mathscr{B}(h)\};\\
\mathscr{B}_n^\Sigma(h_B) =& \{h'_B | h' \in \mathscr{B}(h)\};\\
\mathscr{B}_n^\Sigma(h_C) =& \{h'_C | h' \in \mathscr{B}(h)\};\\
\mathscr{C}_n^\Sigma(h_W) =& \{h'_C | h' \in \mathscr{C}(h)\};\\
\mathscr{C}_n^\Sigma(h_B) =& \{h'_C | h' \in \mathscr{C}(h)\};\\
\mathscr{C}_n^\Sigma(h_C) =& \{h'_C | h' \in \mathscr{C}(h)\};\\
\mathscr{V}_n^\Sigma(p) =& \{(h_W, k) | k \geq n \textit{ and } M, h(k) \models \sigma_W(p, k-n)\} \cup\\
& \{(h_W, k) | k < n \textit{ and } M, h(k) \models p\} \cup\\
& \{(h_B, k) | k \geq n \textit{ and } M, h(k) \models \sigma_B(p, k-n)\} \cup\\
& \{(h_B, k) | k < n \textit{ and } M, h(k) \models p\} \cup\\
& \{(h_C, k) | k \geq n \textit{ and } M, h(k) \models \sigma_C(p, k-n)\} \cup\\
& \{(h_C, k) | k < n \textit{ and } M, h(k) \models p\}.
\end{aligned}
$$

M_n^Σ is obtained by creating three copies of each history of the original model M (a copy for the physical world, a copy for belief, a copy for choice). Moreover, for every atom p and for every $k \in \mathbb{N}$ such that $k \geq n$, the effect of updating model M at the time point n by the event Σ is to assign the truth value of $\sigma_B(p, k-n)$ to the atom p at the time point k of all belief copies of the original histories, to assign the truth value of $\sigma_C(p, k-n)$ to the atom p at the time point k of all choice copies of the original histories, and to assign the truth value of $\sigma_W(p, k-n)$ to the atom p at the time point k of all world copies of the original histories. For example, suppose that $k = n$. Then, the effect of updating model M at the time point n by the event Σ, is to assign the truth value of $\sigma_B(p, 0)$ (resp. $\sigma_C(p, 0)$, resp. $\sigma_W(p, 0)$) to the atom p at the time point n of all belief copies (resp. choice copies, resp. world copies) of the original histories.

For every world copy h_W, at h_W the agent considers possible all belief copies of those histories that he considered possible before the event Σ, and he chooses all choice copies of those histories that he chose before the event Σ.

For every belief copy h_B, at h_B the agent considers possible all belief copies of those histories that he considered possible before the event Σ, and he chooses all choice copies of those histories that he chose before the event Σ.

For every choice copy h_C, at h_C the agent considers possible all choice copies of those histories that he considered possible before the event Σ, and he chooses all choice copies of those histories that he chose before the event Σ.

This construction of the updated model M_n^Σ ensures that the agent is aware that his choices have been changed accordingly so that the properties of positive and negative introspection over the agent's choices (Constraint C2 in Definition 1) are preserved after the occurrence of the event Σ.

Theorem 2. *If* M *is an* $\mathbf{L}$*-model then* M_n^Σ *is an* $\mathbf{L}$*-model.*

Proof. It is just trivial to prove that our operation of model update preserves Constraint C1 in Definition 1. Let us prove that it also preserves Constraint C2.

Assume $h'_B \in \mathscr{B}^{\Sigma}_n(h_W)$ and $h''_C \in \mathscr{C}^{\Sigma}_n(h'_B)$. It follows that $h' \in \mathscr{B}(h)$ and $h'' \in \mathscr{C}(h')$. Then, by constraint C2, we have $h'' \in \mathscr{C}(h)$. Therefore, $h''_C \in \mathscr{C}^{\Sigma}_n(h_W)$. In a similar way we can prove that if $h'_B \in \mathscr{B}^{\Sigma}_n(h_B)$ and $h''_C \in \mathscr{C}^{\Sigma}_n(h'_B)$ then $h''_C \in \mathscr{C}^{\Sigma}_n(h_B)$, and if $h'_C \in \mathscr{B}^{\Sigma}_n(h_C)$ and $h''_C \in \mathscr{C}^{\Sigma}_n(h'_C)$ then $h''_C \in \mathscr{C}^{\Sigma}_n(h_C)$.

Now, assume $h'_C \in \mathscr{C}^{\Sigma}_n(h_W)$ and $h''_B \in \mathscr{B}^{\Sigma}_n(h_W)$. It follows that $h' \in \mathscr{C}(h)$ and $h'' \in \mathscr{B}(h)$. Then, by constraint C2, we have $h' \in \mathscr{C}(h'')$. Therefore, $h'_C \in \mathscr{C}^{\Sigma}_n(h''_B)$. In a similar way we can prove that if $h'_C \in \mathscr{C}^{\Sigma}_n(h_B)$ and $h''_B \in \mathscr{B}^{\Sigma}_n(h_B)$ then $h'_C \in \mathscr{C}^{\Sigma}_n(h''_B)$, and if $h'_C \in \mathscr{C}^{\Sigma}_n(h_C)$ and $h''_C \in \mathscr{B}^{\Sigma}_n(h_C)$ then $h'_C \in \mathscr{C}^{\Sigma}_n(h''_C)$. □

Definition 5 (Truth of L$^+$-formulae). *The satisfaction relation $\models$, between formulae in $\boldsymbol{L}^+$ and pointed models, is defined by the conditions in Definition 2 together with the following three conditions:*

- $M, h(n) \models [\Sigma{:}\textsc{w}]\varphi$ *iff* $M^{\Sigma}_n, h_W(n{+}1) \models \varphi$;
- $M, h(n) \models [\Sigma{:}\textsc{b}]\varphi$ *iff* $M^{\Sigma}_n, h_B(n{+}1) \models \varphi$;
- $M, h(n) \models [\Sigma{:}\textsc{c}]\varphi$ *iff* $M^{\Sigma}_n, h_C(n{+}1) \models \varphi$.

Note that, according to the previous definition, the occurrence of the event Σ takes time. That is, an event Σ is a transition from a time point n along a history h of a model M to the *successor* of time point n along the world copy (or belief copy, or choice copy) of history h in the updated model M^{Σ}_n.

3.3 Axiomatization

We have reduction axioms for the three operators $[\Sigma{:}\textsc{w}]$, $[\Sigma{:}\textsc{b}]$ and $[\Sigma{:}\textsc{c}]$. They are called reduction axioms because, read from left to right, they reduce the complexity of those operators in a formula.

Theorem 3. *Suppose $\Sigma = (\sigma_B,\sigma_C,\sigma_W)$. Then, the following schemata are valid in $\boldsymbol{L}^+$:*

R1a.	$[\Sigma{:}\textsc{w}]p \leftrightarrow \sigma_W(p,1)$	***R4a.***	$[\Sigma{:}\textsc{w}]\bigcirc\varphi \leftrightarrow \bigcirc[\Uparrow\Sigma{:}\textsc{w}]\varphi$
R1b.	$[\Sigma{:}\textsc{b}]p \leftrightarrow \sigma_B(p,1)$	***R4b.***	$[\Sigma{:}\textsc{b}]\bigcirc\varphi \leftrightarrow \bigcirc[\Uparrow\Sigma{:}\textsc{b}]\varphi$
R1c.	$[\Sigma{:}\textsc{c}]p \leftrightarrow \sigma_C(p,1)$	***R4c.***	$[\Sigma{:}\textsc{c}]\bigcirc\varphi \leftrightarrow \bigcirc[\Uparrow\Sigma{:}\textsc{c}]\varphi$
R2a.	$[\Sigma{:}\textsc{w}]\neg\varphi \leftrightarrow \neg[\Sigma{:}\textsc{w}]\varphi$	***R5a.***	$[\Sigma{:}\textsc{w}][\mathsf{B}]\varphi \leftrightarrow [\mathsf{B}][\Sigma{:}\textsc{b}]\varphi$
R2b.	$[\Sigma{:}\textsc{b}]\neg\varphi \leftrightarrow \neg[\Sigma{:}\textsc{b}]\varphi$	***R5b.***	$[\Sigma{:}\textsc{b}][\mathsf{B}]\varphi \leftrightarrow [\mathsf{B}][\Sigma{:}\textsc{b}]\varphi$
R2c.	$[\Sigma{:}\textsc{c}]\neg\varphi \leftrightarrow \neg[\Sigma{:}\textsc{c}]\varphi$	***R5c.***	$[\Sigma{:}\textsc{c}][\mathsf{B}]\varphi \leftrightarrow [\mathsf{B}][\Sigma{:}\textsc{c}]\varphi$
R3a.	$[\Sigma{:}\textsc{w}](\varphi \wedge \psi) \leftrightarrow ([\Sigma{:}\textsc{w}]\varphi \wedge [\Sigma{:}\textsc{w}]\psi)$	***R6a.***	$[\Sigma{:}\textsc{w}][\mathsf{C}]\varphi \leftrightarrow [\mathsf{C}][\Sigma{:}\textsc{c}]\varphi$
R3b.	$[\Sigma{:}\textsc{b}](\varphi \wedge \psi) \leftrightarrow ([\Sigma{:}\textsc{b}]\varphi \wedge [\Sigma{:}\textsc{b}]\psi)$	***R6b.***	$[\Sigma{:}\textsc{b}][\mathsf{C}]\varphi \leftrightarrow [\mathsf{C}][\Sigma{:}\textsc{c}]\varphi$
R3c.	$[\Sigma{:}\textsc{c}](\varphi \wedge \psi) \leftrightarrow ([\Sigma{:}\textsc{c}]\varphi \wedge [\Sigma{:}\textsc{c}]\psi)$	***R6c.***	$[\Sigma{:}\textsc{c}][\mathsf{C}]\varphi \leftrightarrow [\mathsf{C}][\Sigma{:}\textsc{c}]\varphi$

Proof. We prove only **R4a** as an example.
$M, h(n) \models [\Sigma{:}\textsc{w}]\bigcirc\varphi$
IFF $M^{\Sigma}_n, h_W(n{+}1) \models \bigcirc\varphi$
IFF $M^{\Sigma}_n, h_W((n{+}1){+}1) \models \varphi$
IFF $M^{\Uparrow\Sigma}_{n+1}, h_W((n{+}1){+}1) \models \varphi$ (because $M^{\Sigma}_n = M^{\Uparrow\Sigma}_{n+1}$)
IFF $M, h(n{+}1) \models [\Uparrow\Sigma{:}\textsc{w}]\varphi$
IFF $M, h(n) \models \bigcirc[\Uparrow\Sigma{:}\textsc{w}]\varphi$. □

Theorem 4. *The logic $\mathbf{L}^+$ is completely axiomatized by principles in Fig. 1 together with the schemata of Theorem 3 and the rule of replacement of proved equivalence.*

Proof. Using the reduction axioms **R1a**-**R6c** in Theorem 3, and the rule of replacement of proved equivalence, every $\mathbf{L}^+$ formula can be reduced to an equivalent $\mathbf{L}$ formula. Hence, the completeness of $\mathbf{L}^+$ is a straightforward consequence of Theorem 1. □

In the rest of the paper we write $\vdash_{\mathbf{L}^+} \varphi$ if φ is a $\mathbf{L}^+$-theorem.

3.4 Discussion

The present approach offers two different temporal perspectives on world evolution, where the world includes both the physical world and the mental world (*i.e.* the agent's beliefs and choices).

In order to describe the *passive (or inertial) evolution of the world* (*i.e.* how the world evolves over time when there are no occurrences of events which affect the physical world and the agent's beliefs and choices), we use the *next* operator $\bigcirc$ in the static framework $\mathbf{L}$. Note that the operator $\bigcirc$ corresponds in the semantics to a transition from a state in the current model M to the *unique* successor state in the same model M.

In order to describe the *active evolution of the world* (*i.e.* how the world evolves over time when there are occurrences of events which affect the physical world and the agent's beliefs and choices), we use the dynamic operators $[\Sigma{:}\textsc{w}]$, $[\Sigma{:}\textsc{b}]$ and $[\Sigma{:}\textsc{c}]$ in the dynamic framework $\mathbf{L}^+$. We consider all possible transitions from the current model M to a new model which corresponds to the update of M through an event Σ affecting the physical world and the agent's beliefs and choices.

In other terms, in order to describe the *active evolution of the world* a branching time perspective is adopted in our approach. That is, we suppose that the world might *actively* evolve in many different ways depending on the event Σ which occurs and which affects it. On the contrary, in order to describe the *inertial evolution of the world* a linear time perspective is adopted in our approach, that is, we suppose that the world *passively/inertially* evolves in a deterministic way.

4 Intention and Plan Dynamics in $\mathbf{L}^+$

Two basic operations on an agent's intentions can be defined in $\mathbf{L}^+$: the operation of *generating* an intention to do an action α n steps from now, noted $gen(\alpha,n)$; and the operation of *reconsidering* (or *erasing*) an intention to do an action α n steps from now, noted $rec(\alpha,n)$. These two operations are defined as follows by means of choice assignments: [2]

$$gen(\alpha,n) \overset{\text{def}}{=} (\alpha,n) \overset{C}{\mapsto} \top$$

$$rec(\alpha,n) \overset{\text{def}}{=} (\alpha,n) \overset{C}{\mapsto} \bot$$

[2] The generalization to conjunctive intentions can be done with simultaneous assignments. For instance, operation $gen(\alpha \wedge \beta,n) \overset{\text{def}}{=} (\alpha,n) \overset{C}{\mapsto} \top, (\beta,n) \overset{C}{\mapsto} \top$, *i.e.* a partial local assignment function with domain $\{(\alpha,n),(\beta,n)\}$.

The following are $\mathbf{L}^+$-theorems which highlight some interesting properties of intention generation and intention reconsideration. For every $n, m \in \mathbb{N}$ and for every $\alpha, \beta \in ACT$ we have:

(1) $\vdash_{\mathbf{L}^+} [(\emptyset,\{gen(\alpha,n{+}1)\},\emptyset){:}w]\mathtt{I}^n(\alpha)$

(2) $\vdash_{\mathbf{L}^+} [(\emptyset,\{rec(\alpha,n{+}1)\},\emptyset){:}w]\neg\mathtt{I}^n(\alpha)$

(3) $\vdash_{\mathbf{L}^+} [(\emptyset,\{gen(\alpha,n{+}1)\},\emptyset){:}c]\bigcirc^n\alpha$

(4) $\vdash_{\mathbf{L}^+} [(\emptyset,\{rec(\alpha,n{+}1)\},\emptyset){:}c]\neg\bigcirc^n\alpha$

(5) $\vdash_{\mathbf{L}^+} \neg\mathtt{I}^m(\beta) \rightarrow [(\emptyset,\{gen(\alpha,n)\},\emptyset){:}w]\neg\mathtt{I}^{m-1}(\beta)$
if $\alpha \neq \beta$ or $m \neq n$

(6) $\vdash_{\mathbf{L}^+} \mathtt{I}^m(\beta) \rightarrow [(\emptyset,\{rec(\alpha,n)\},\emptyset){:}w]\mathtt{I}^{m-1}(\beta)$
if $\alpha \neq \beta$ or $m \neq n$

Proof. We prove theorem (1) as an example. Formula $[(\emptyset,\{gen(\alpha,n{+}1)\},\emptyset){:}w]\mathtt{I}^n(\alpha)$ is equivalent to $[\mathtt{C}][(\emptyset,\{gen(\alpha,n{+}1)\},\emptyset){:}c]\bigcirc^n\alpha$ (by **R6a** and rule of replacement of proved equivalences). The latter is equivalent to $[\mathtt{C}]\bigcirc^n[(\emptyset,\{gen(\alpha,1)\},\emptyset){:}c]\alpha$ (by repeated application of **R4c** and rule of replacement of proved equivalence) which in turn is equivalent to $[\mathtt{C}]\bigcirc^n\top$ (by **R1c** and rule of replacement of proved equivalence). The latter is equivalent to $\top$. □

According to theorem (1), after generating the intention to do α $n{+}1$ steps from now, in the physical world the agent intends to do α n steps from now. According to theorem (2), after reconsidering the intention to do α $n{+}1$ steps from now, in the physical world the agent does not intend to do α n steps from now. In Definition 5 we have supposed that the occurrence of a local assignment takes time (one time unit). Consequently, also the processes of generating/reconsidering an intention takes time. This is the reason why, as stated by theorems (1) and (2), the process of generating/reconsidering the intention to do α $n{+}1$ steps from now generates/reconsiders an intention to α n steps from now, and not an intention to do α $n{+}1$ steps from now.

Note that the two processes of intention generation and intention reconsideration comply with temporal precedence, that is, the process of reconsidering a certain intention cancels the effects of a previous process of generating the same intention, and the process of generating a certain intention cancels the effects of a previous process of reconsidering the same intention. More formally, by theorems (1) and (2), we have:

$$\vdash_{\mathbf{L}^+} [(\emptyset,\{gen(\alpha,n{+}2)\},\emptyset){:}w][(\emptyset,\{rec(\alpha,n{+}1)\},\emptyset){:}w]\neg\mathtt{I}^n(\alpha)$$
$$\vdash_{\mathbf{L}^+} [(\emptyset,\{rec(\alpha,n{+}2)\},\emptyset){:}w][(\emptyset,\{gen(\alpha,n{+}1)\},\emptyset){:}w]\mathtt{I}^n(\alpha)$$

Theorems (3) and (4) express the corresponding effects of the processes of intention generation and of intention reconsideration in the context of the agent's choices: after generating (resp. reconsidering) the intention to do α $n{+}1$ steps from now, in the context of the agent's choices it is the case that the agent will perform (resp. will not perform) action α n steps from now.

Theorems (5) and (6) express that the operations of intention generation and of intention reconsideration are characterized by partial modifications of an agent's plan. That is, the process of generating/reconsidering a plan does not affect the other plans of the agent: if α and β are different actions or m and n are different, and the agent intends (resp. does not intend) to do β m steps from now then, after reconsidering (resp. generating) the intention do α n steps from now, the agent will intend (resp. not intend) to do β $m-1$ steps from now.

Intention generation ($gen(\alpha,n)$) and reconsideration ($rec(\alpha,n)$) are mental events which have to be distinguished from the processes of starting an action (or trying to do an action) and stopping an action which are events operating on the physical world. The latter are defined by means of world assignments as follows:

$$start(\alpha) \overset{\text{def}}{=} (\alpha,1) \overset{W}{\mapsto} \top$$

$$stop(\alpha) \overset{\text{def}}{=} (\alpha,1) \overset{W}{\mapsto} \bot$$

The following are two $\mathbf{L}^+$-theorems which highlight the basic properties of the processes of starting an action and stopping an action. For every $\alpha \in ACT$ we have:

(7) $\vdash_{\mathbf{L}^+} [(\emptyset,\emptyset,\{start(\alpha)\}){:}w]\alpha$

(8) $\vdash_{\mathbf{L}^+} [(\emptyset,\emptyset,\{stop(\alpha)\}){:}w]\neg\alpha$

According to theorem (7), after starting action α, the agent performs action α. According to theorem (8), after stopping action α, the agent does not perform action α.

5 Application

Before concluding, we illustrate through an example how $\mathbf{L}^+$ can be concretely used to model intention dynamics.

Executability of intention generation. We denote with $\langle\langle Gen(\alpha,n)\rangle\rangle\varphi$ the fact 'it is possible that the agent will generate the intention to do action α n steps from now, and φ will be true afterwards'. Consequently, $\langle\langle Gen(\alpha,n)\rangle\rangle\top$ just means 'the agent will possibly generate the intention to do action α n steps from now'. The construction $\langle\langle Gen(\alpha,n)\rangle\rangle\varphi$ is defined as follows.

$$\langle\langle Gen(\alpha,n)\rangle\rangle\varphi \overset{\text{def}}{=} \neg\mathtt{I}^n(\alpha) \wedge [\mathtt{B}]\bigcirc^n good_\alpha \wedge \\ [(\emptyset,\{gen(\alpha,n)\},\emptyset){:}w]\varphi$$

According to this definition, the *executability* of the process of generating the intention to do α n steps from now is determined by two conditions: the agent does not have already this intention (*i.e.* $\neg\mathtt{I}^n(\alpha)$) and he believes that, n steps from now, doing action α will be something good for him (*i.e.* $[\mathtt{B}]\bigcirc^n good_\alpha$), where $good_\alpha$ is a special atom in ATM^{Fact} expressing that 'performing action α is good for the agent'. Indeed, $\langle\langle Gen(\alpha,n)\rangle\rangle\top$ and $\neg\mathtt{I}^n(\alpha) \wedge [\mathtt{B}]\bigcirc^n good_\alpha$ are logically equivalent. Note that the condition $[\mathtt{B}]\bigcirc^n good_\alpha$ corresponds to the notion of *reasons for intending* or *reasons for acting*. This notion has been extensively studied in the philosophical literature on action and intention (see, *e.g.* [28,20]). A reason for intending is a belief that the agent uses as premise of a practical argument (*viz.* the argument that concludes in an intention).

Executability of intention reconsideration. The notion of executability of the process of intention reconsideration is defined in a similar way as follows:

$$\langle\langle Rec(\alpha,n)\rangle\rangle\varphi \overset{\mathtt{def}}{=} \mathtt{I}^n(\alpha) \wedge [\mathtt{B}]\bigcirc^n \neg good_\alpha \wedge$$
$$[(\emptyset,\{rec(\alpha,n)\},\emptyset):w]\varphi$$

$\langle\langle Rec(\alpha,n)\rangle\rangle\varphi$ denotes the fact 'it is possible that the agent will reconsider his intention to do action α n steps from now, and φ will be true afterwards'. Consequently, $\langle\langle Gen(\alpha,n)\rangle\rangle\top$ just means 'the agent will possibly reconsider his intention to do action α n steps from now'. According to this definition, the *executability* of the process of reconsidering the intention to do α n steps from now is determined by two conditions: (1) the agent has this intention (*i.e.* $\mathtt{I}^n(\alpha)$) and (2) he believes that, n steps from now, doing action α will be something bad for him (*i.e.* $[\mathtt{B}]\bigcirc^n\neg good_\alpha$). [3] Indeed, $\langle\langle Rec(\alpha,n)\rangle\rangle\top$ and $\mathtt{I}^n(\alpha) \wedge [\mathtt{B}]\bigcirc^n\neg good_\alpha$ are logically equivalent.

Executability of action. The notion of executability of the process of starting an action is defined as follows:

$$\langle\langle Start(\alpha)\rangle\rangle\varphi \overset{\mathtt{def}}{=} \mathtt{I}^1(\alpha) \wedge [\mathtt{B}]\bigcirc good_\alpha \wedge$$
$$[(\emptyset,\emptyset,\{start(\alpha)\}):w]\varphi$$

$\langle\langle Start(\alpha)\rangle\rangle\varphi$ denotes the fact 'it is possible that the agent will start action α, and φ will be true afterwards'. Consequently, $\langle\langle Gen(\alpha,n)\rangle\rangle\top$ just means 'the agent will possibly start action α'. According to this definition, the executability of the process of starting action α is determined by two conditions: (1) the agent has intention to perform action α in the next step ($\mathtt{I}^1(\alpha)$) and (2) he believes that, in the next state, doing action α will be something good for him (*i.e.* $[\mathtt{B}]\bigcirc good_\alpha$). Indeed, $\langle\langle Start(\alpha)\rangle\rangle\top$ and $\mathtt{I}^1(\alpha)\wedge[\mathtt{B}]\bigcirc good_\alpha$ are logically equivalent.

An example. We suppose that the agent wants to reach a certain place called *Utopia* in n steps from now, with $n \geq 4$. He can go to *Utopia* either by train or by car. That is, $ATM^{Act} = \{train, car\}$. The agent has decided to go by train. Thus, he has the intention to go to *Utopia* by train and does not have the intention to go to *Utopia* by car:

H. $\mathtt{I}^n(train) \wedge \neg\mathtt{I}^n(car)$.

Now, suppose the agent is informed that there is a train strike that day and, n steps from now, there will be no train going to *Utopia*. Thus, the agent learns that going to *Utopia* by train is a bad solution, whereas going to *Utopia* by car is a good solution. How are the agent's intentions and plans affected by this new information? The following theorem clarifies this point.

(9)
$$\vdash_{\mathbf{L}^+} \mathbf{H} \rightarrow [(\{(good_{train},n) \overset{B}{\mapsto} \bot, (good_{car},n) \overset{B}{\mapsto} \top\},\emptyset,\emptyset):w]$$
$$\langle\langle Rec(train,n{-}1)\rangle\rangle\langle\langle Gen(car,n{-}2)\rangle\rangle$$
$$\bigcirc^{n-4}\langle\langle Start(car)\rangle\rangle car$$

[3] We here suppose that the agent believes that doing α will be something *bad* for him if and only if, he believes that doing α will be something *not good* for him.

According to theorem (9), hypothesis **H** ensures that, after having learnt that going to *Utopia* by train is a bad solution and going to *Utopia* by car is a good solution:

- the agent will possibly reconsider its intention to go to *Utopia* by train $n-1$ steps from now and,
- after that, it will possibly generate the intention to go to *Utopia* by car $n-2$ steps from now and,
- $n-4$ steps later, if the world will evolve passively/inertially, the agent will possibly start to go to *Utopia* by car and,
- as a consequence, he will go to *Utopia* by car.

Thus, theorem (9) shows that $\mathbf{L}^+$ concretely models intention dynamics in this example.

6 Related Work and Perspectives

Assignments were studied before in the literature on logic for information dynamics. However they were only applied to the dynamics of belief and knowledge [12,3], and there is still no application of this notion to the theory of intention. Furthermore, previous works in the area of Dynamic Epistemic Logic (DEL) [11] have only focused on assignments at a specific moment in time, and have not considered delayed assignments, that can operate locally, *i.e.* on specific future points in a model. In this paper we have shown that local assignments are well-suited to model intention and plan dynamics.

It has also to be noted that complex assignments restricted to $n = 0$ correspond to a three-event BMS action model [1]. Given a future point n in time, one can in principle again construct an action model from the composition of complex assignments and next operators in a sequence of length n (where the next operators correspond to 'clock ticks', so-called 'nothing happens' action models).

In [4,19] a logic of knowledge and preference dynamics is provided. In van Benthem & Liu's approach knowledge dynamics are modeled by means of announcements (or updates), whereas preference dynamics are modeled by means of operations on accessibility relations called upgrades. We think that local assignments, rather than announcements and upgrades, are more suited to model intention dynamics. Indeed, intention dynamics are obtained by partial modifications of an agent's plan and not necessarily by global modifications, and local assignments are a natural candidate to formalize these kinds of operations. This aspect of intention dynamics has been discussed in Section 3.3.

In [16], van der Hoek *et al.* present a formal model of intention revision strongly inspired by [9]. In this model an agent's mental state (which includes the agent's beliefs, desires and intentions) changes because the agent has made observations of his environment. In particular, observations cause change in beliefs and, indirectly, may produce change in the agent's intentions. The treatment of beliefs, desires, intentions and their dynamics proposed by van der Hoek *et al.* is rather syntactical (*i.e.* an agent's beliefs, desires and intentions are just sets of sentences and there are no modalities for these mental attitudes interpreted by means of model structures). This is the main difference with our approach whose objective is to provide a model-theoretic semantics of intentions and intention dynamics.

In [26], Shoham discusses the interaction of belief revision and intention revision. He treats belief revision more in the traditional AGM sense than in the dynamic epistemic modal sense. The paper has the character of a requirements analysis. A prior paper on belief and intention interaction in the AGM tradition, however without intention dynamics, is [8]. Work by Icard, Pacuit and Shoham [17] employs a dynamic modal approach for intention revision.

Rodenhäuser [24] associates an intention with a protocol, a sequence of planned actions that may also involve factual change—but not the local change as in our case; he also does not employ choice modalities.

Directions for future research are manifold. In this paper we only considered the single-agent case. We plan to extend our approach to the multi-agent setting in which agents can act in parallel and communicate their choices, beliefs and intentions to other agents. As anticipated in Section 2.3, we also plan to refine our logical framework by adding more interaction principles between the belief operator and the choice operator.

References

1. Baltag, A., Moss, L., Solecki, S.: The logic of public announcements, common knowledge and private suspicions. In: Proceedings of TARK 1998, pp. 43–56. Morgan Kaufmann, San Francisco (1998)
2. Bell, J., Huang, Z.: Dynamic goal hierarchies. Intelligent Agent Systems Theoretical and Practical Issues, 88–103 (1997)
3. van Benthem, J., van Eijck, J., Kooi, B.: Logics of communication and change. Information and Computation 204(11), 1620–1662 (2006)
4. van Benthem, J., Liu, F.: Dynamic logic of preference upgrade. Journal of Applied Non-Classical Logics 17(2), 157–182 (2007)
5. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
6. Bratman, M.: Intentions, plans, and practical reason. Harvard University Press, Cambridge (1987)
7. Broersen, J., Dastani, M., Hulstijn, J., van der Torre, L.: Goal generation in the BOID architecture. Cognitive Science Quarterly 2(3-4), 431–450 (2002); special issue on Desires, goals, intentions, and values: Computational architectures
8. Cleaver, T.W., Sattar, A.: Intention guided belief revision. In: Proceedings of AAAI 2007, pp. 36–41. AAAI Press, Menlo Park (2007)
9. Cohen, P.R., Levesque, H.J.: Intention is choice with commitment. Artificial Intelligence 42, 213–261 (1990)
10. Dignum, F., Kinny, D.: From desires, obligations and norms to goals. Cognitive Science Quarterly 2(3-4), 407–430 (2002)
11. van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic Epistemic Logic. Synthese Library Series, vol. 337. Springer, Heidelberg (2007)
12. van Ditmarsch, H., Kooi, B.: Semantic results for ontic and epistemic change. In: Logic and the Foundations of Game and Decision Theory (LOFT 7)., pp. 87–117. Texts in Logic and Games, Amsterdam University Press (2008)
13. Dunin-Keplicz, B., Verbrugge, R.: Collective intentions. Fundamenta Informaticae 51(3), 271–295 (2002)
14. Fagin, R., Halpern, J., Moses, Y., Vardi, M.: Reasoning about Knowledge. MIT Press, Cambridge (1995)

15. Georgeff, M.P., Rao, A.S.: The semantics of intention maintenance for rational agents. In: Proceedings of IJCAI 1995, pp. 704–710 (1995)
16. Van der Hoek, W., Jamroga, W., Wooldridge, M.: Towards a theory of intention revision. Synthese 155(2), 265–290 (2007)
17. Icard, T., Pacuit, E., Shoham, Y.: Joint revision of beliefs and intention. In: Proceedings of KR 2010, pp. 572–574 (2010)
18. Khan, S.M., Lesperance, Y.: A logical framework for prioritized goal change. In: Proceedings of AAMAS 2010, pp. 283–290 (2010)
19. Liu, F.: Changing for the Better: Preference Dynamics and Agent Diversity. Ph.D. thesis, University of Amsterdam (2008)
20. Lorini, E., Herzig, A.: A logic of intention and attempt. Synthese 163(1), 45–77 (2008)
21. Lorini, E., van Ditmarsch, H., de Lima, T.: Logical model of intention and plan dynamics. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) Proceedings of ECAI 2010, pp. 1075–1076. IOS Press, Amsterdam (2010)
22. Meyer, J.J.C., van der Hoek, W., van Linder, B.: A logical approach to the dynamics of commitments. Artificial Intelligence 113(1-2), 1–40 (1999)
23. Rao, A.S., Georgeff, M.P.: Modelling rational agents within a BDI-architecture. In: Proceedings of KR 1991, pp. 473–484. Morgan Kaufmann, San Francisco (1991)
24. Rodenhäuser, B.: Intentions in interaction. In: Proceedings of LOFT 2010 (2010)
25. Sahlqvist, H.: Completeness and correspondence in the first and second order semantics for modal logics. In: Proceedings of the 3rd Scandinavian Logic Symposium 1973, Studies in Logic, vol. (82) (1975)
26. Shoham, Y.: Logics of intention and the database perspective. Journal of Philosophical Logic 38(6), 633–647 (2009)
27. Singh, M., Asher, N.: A logic of intentions and beliefs. Journal of Philosophical Logic 22, 513–544 (1993)
28. Von Wright, G.H.: On so-called practical inference. The Philosophical Review 15, 39–53 (1972)
29. Wooldridge, M.: Reasoning about rational agents. MIT Press, Cambridge (2000)

Author Index