Matteo Baldoni and Ulle Endriss (eds.)

Declarative Agent Languages and Technologies

Fourth International Workshop, DALT 2006
Hakodate, Japan, May 8th, 2006
Workshop Notes

DALT 2006 Home Page:
`http://staff.science.uva.nl/~ulle/DALT-2006/`

Preface

The workshop on Declarative Agent Languages and Technologies (DALT), in its *fourth edition* this year, is a well-established forum for researchers interested in sharing their experiences in combining declarative and formal approaches with engineering and technology aspects of agents and multiagent systems. Building complex agent systems calls for models and technologies that ensure predictability, allow for the verification of properties, and guarantee flexibility. Developing technologies that can satisfy these requirements still poses an important and difficult challenge. Here, declarative approaches have the potential of offering solutions that satisfy the needs for both specifying and developing multiagent systems. Moreover, they are gaining more and more attention in important application areas such as the semantic web, web services, security, and electronic contracting.

DALT 2006 is being held as a satellite workshop of AAMAS 2006, the 5th International Joint Conference on Autonomous Agents and Multiagent Systems, in Hakodate, Japan. Following the success of DALT 2003 in Melbourne (LNAI 2990), DALT 2004 in New York (LNAI 3476), and DALT 2005 in Utrecht (LNAI 3904), DALT will again aim at providing a discussion forum to both (i) support the transfer of declarative paradigms and techniques to the broader community of agent researchers and practitioners, and (ii) to bring the issue of designing complex agent systems to the attention of researchers working on declarative languages and technologies.

This volume containts the twelve papers that have been selected by the Programme Committee for presentation at the workshop. Each paper received at least three reviews in order to supply the authors with a rich feedback that could stimulate the research as well as foster the discussion. In addition to these presentations, Munindar P. Singh from North Carolina State University will be giving an invited talk on a declarative approach to instantiating business protocols.

We would like to thank all authors for their contributions, the members of the Steering Committee for the precious suggestions and support, and the members of the Programme Committee for the excellent work during the reviewing phase.

March 19th, 2006

Matteo Baldoni
Ulle Endriss

Workshop Organisers

Matteo Baldoni	University of Torino, Italy
Ulle Endriss	University of Amsterdam, The Netherlands

Programme Committee

Marco Alberti	University of Ferrara, Italy
Natasha Alechina	University of Nottingham, UK
Grigoris Antoniou	University of Crete, Greece
Matteo Baldoni	University of Torino, Italy, *Co-chair*
Cristina Baroglio	University of Torino, Italy
Rafael Bordini	University of Durham, UK
Keith Clark	Imperial College London, UK
Ulle Endriss	University of Amsterdam, The Netherlands, *Co-chair*
Benjamin Hirsch	Technical University Berlin, Germany
Shinichi Honiden	National Institute of Informatics, Japan
John Lloyd	Australian National University, Australia
Viviana Mascardi	University of Genova, Italy
John-Jules Ch. Meyer	Utrecht University, The Netherlands
Enrico Pontelli	New Mexico State University, USA
Birna van Riemsdijk	University of Utrecht, The Netherlands
Chiaki Sakama	Wakayama University, Japan
Wamberto Vasconcelos	University of Aberdeen, UK
Christopher Walton	University of Edinburgh, UK
Michael Winikoff	RMIT University, Melbourne, Australia

Steering Committee

João Leite	New University of Lisbon, Portugal
Andrea Omicini	University of Bologna-Cesena, Italy
Leon Sterling	University of Melbourne, Australia
Paolo Torroni	University of Bologna, Italy
Pınar Yolum	Bogazici University, Turkey

Additional Reviewers

Giovanni Casella
Valentina Cordì
John Knottenbelt

Sponsoring Institutions

Matteo Baldoni has partially been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://rewerse.net).

Table of Contents

Automating Belief Revision for AgentSpeak

Natasha Alechina[1], Rafael H. Bordini[2], Jomi F. Hübner[3],
Mark Jago[1], and Brian Logan[1]

[1] School of Computer Science
University of Nottingham
Nottingham, UK
{nza,mtw,bsl}@cs.nott.ac.uk

[2] University of Durham
Dept. of Computer Science
Durham, UK
r.bordini@durham.ac.uk

[3] Univ. Regional de Blumenau
Dept. Sistemas e Computação
Blumenau, SC, Brazil
jomi@inf.furb.br

Abstract. The AgentSpeak agent-oriented programming language has recently been extended with various new features, such as speech-act based communication, internal belief additions, and support for reasoning with ontological knowledge, which imply the need for belief revision within an AgentSpeak agent. In this paper, we show how a polynomial-time belief-revision algorithm can be incorporated into the ***Jason*** AgentSpeak interpreter by making use of ***Jason***'s language constructs and customisation features. This is one of the first attempts to include automatic belief revision within an interpreter for a practical agent programming language.

1 Introduction

After almost a decade of work on abstract programming languages for multi-agent systems, practical multi-agent platforms based on these languages are now beginning to emerge. One example of a well-known agent language that has evolved to the point of being sufficiently practical for widespread use is AgentSpeak, and in particular its implementation in ***Jason*** [7]. A number of extensions to AgentSpeak have been reported in the literature and incorporated into ***Jason***. Some of these new features, such as speech-act based communication, internal belief additions, and support for reasoning with ontological knowledge, have led to a greater need for *belief revision* as part of an agent's reasoning cycle. However, in common with other mature agent-oriented programming languages [5], ***Jason*** does not currently provide automatic support for belief revision. The current implementation provides a simple form of belief *update*, which can be customised for particular applications. However, the problem of belief-base consistency has, so far, remained the responsibility of the programmer.

The lack of support for belief revision in practical agent programming languages is understandable, given that known belief revision algorithms have high computational complexity bounds. However recent work by Alechina et al. [2] has changed this picture. By making simplifying assumptions, which nevertheless are quite realistic for agent-oriented programming languages, they were able to produce a polynomial-time belief-revision algorithm, which is also theoretically well-motived, in the sense of producing revisions that conform to a generally accepted set of postulates characterising *rational* belief revision. In this paper, we show how this work can be incorporated into the ***Jason*** AgentSpeak interpreter by making use of ***Jason***'s language constructs and customisation features. This is one of the first attempts to include automatic belief revision within an interpreter for a practical agent programming language. Some initial considerations on belief revision in an *abstract* programming language appeared, for example, in [23]. In an approach similar to ours, [10] sketch how the `Go!` programming language can be extended with a consistency maintenance system which can be used by an agent whose beliefs are constrained by a formal ontology to decide which beliefs to remove in order to restore consistency.

The remainder of the paper is organised as follows. In Sections 2 and 3 we give a brief overview of AgentSpeak programming and its implementation in ***Jason***. In Section 4, we state our desiderata for belief revision in AgentSpeak, and in Section 5 we summarise the main points of the algorithm first introduced in [2]. We then discuss the integration of the belief revision algorithm into ***Jason*** in Section 6, while Section 7 gives a simple example which illustrates the importance of belief revision in practical programming of multi-agent systems. Finally, we discuss conclusions and future work.

2 AgentSpeak

The AgentSpeak(L) programming language was introduced in [21]. It is based on logic programming and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to the implementation of *intelligent* or *rational* agents [26], and a number of commercial applications have been developed using this approach.

An AgentSpeak agent is defined by a set of *beliefs* giving the initial state of the agent's *belief base*, which is a set of ground (first-order) atomic formulæ, and a set of plans which form its *plan library*. An AgentSpeak plan has a *head* which consists of a triggering event (specifying the events for which that plan is *relevant*), and a conjunction of belief literals representing a *context*. The conjunction of literals in the context must be a logical consequence of that agent's current beliefs if the plan is to be considered *applicable* when the triggering event happens (only applicable plans can be chosen for execution). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered. *Basic actions* represent the atomic operations the agent can perform so as to change the environment. Such actions are also written as atomic formulæ, but using a set of *action symbols* rather than predicate symbols. AgentSpeak distinguishes two types of *goals*: achievement goals and test goals. Achievement goals are formed by an atomic formulæ prefixed with the '**`!`**' operator, while test goals are prefixed with the '**`?`**' operator. An *achievement goal* states that the agent wants to achieve a state of the world where the associated atomic formulæ is true. A *test goal* states that the agent wants to test whether the associated atomic formulæ is (or can be unified with) one of its beliefs.

An AgentSpeak agent is a *reactive planning system*. Plans are triggered by the *addition* (‘**+**’) or *deletion* (‘**-**’) of beliefs due to perception of the environment, or to the addition or deletion of goals as a result of the execution of plans triggered by previous events.

A simple example of an AgentSpeak program for a Mars robot is given in Figure 1. The robot is instructed to be especially attentive to “green patches” on rocks it observes while roving on Mars. The AgentSpeak program consists of three plans. The first plan says that whenever the robot perceives a green patch on a certain rock (a belief addition), it should try and examine that particular rock. However this plan can only be used (i.e., it is only applicable) if the robot’s batteries are not too low. To examine the rock, the robot must retrieve, from its belief base, the coordinates it has associated with that rock (this is the reason for the test goal in the beginning of the plan’s body), then achieve the goal of traversing to those coordinates and, once there, examining the rock. Recall that each of these achievement goals will trigger the execution of some other plan.

```
+green_patch(Rock) :
   not battery_charge(low) <-
      ?location(Rock,Coordinates);
      !traverse(Coordinates);
      !examine(Rock).

+!traverse(Coords) :
   safe_path(Coords) <-
      move_towards(Coords).

+!traverse(Coords) :
   not safe_path(Coords) <-
      ...
```

Fig. 1. Examples of AgentSpeak Plans for a Mars Rover

The two other plans (note the last one is only an excerpt) provide alternative courses of action that the rover should take to achieve a goal of traversing towards some given coordinates. Which course of action is selected depends on its beliefs about the environment at the time the goal-addition event is handled. If the rover believes that there is a safe path in the direction to be traversed, then all it has to do is to take the action of moving towards those coordinates (this is a basic action which allows the rover to effect changes in its environment, in this case physically moving itself). The alternative plan (not shown here) provides an alternative means for the agent to reach the rock when the direct path is unsafe.

3 *Jason*

The ***Jason*** interpreter implements the operational semantics of AgentSpeak as given in, e.g., [8]. ***Jason*** [4] is written in Java, and its IDE supports the development and execution of distributed multi-agent systems [6]. Some of the features of ***Jason*** are:

- speech-act based inter-agent communication (and annotation of beliefs with information sources);
- annotations on plan labels, which can be used by elaborate (e.g., decision-theoretic) selection functions;
- the possibility to run a multi-agent system distributed over a network (using SACI or some other middleware);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- straightforward extensibility (and use of legacy code) by means of user-defined "internal actions";
- clear notion of *multi-agent environments*, which can be implemented in Java (this can be a simulation of a real environment, e.g., for testing purposes before the system is actually deployed).

3.1 Extensions to AgentSpeak

Recent work appearing in the literature has made important additions to AgentSpeak, which have also been (or are in the process of being) implemented in ***Jason***. Below we briefly discuss some of these features, focusing on those that have particular implications for belief revision.

Belief additions One of the earliest extensions of the AgentSpeak language is one of the most important from the point of view of belief revision. From the initial work on AgentSpeak, experience showed that it was often the case that the execution of some plans could be greatly facilitated by allowing a plan instance being executed to add derived beliefs to the agent's belief base. A formula such as **+**bl in the body of a plan, has the effect of adding the belief literal bl to the belief base. Together with the ability to exchange plans with other agents (see below), such derived beliefs can result in the agent's belief base becoming inconsistent (i.e., both b and **~**b are in the belief base, for some belief b)[5]. Unless the programmer deliberately intends to make use of paraconsistency, this is clearly undesirable, yet it is *not* currently checked or handled by ***Jason*** automatically.

Speech-act based communication and plan exchange Another important addition, first proposed in [16], is the extension of the AgentSpeak operational semantics to allow speech-act based communication among AgentSpeak agents. That work gave semantics

[4] ***Jason*** is *Open Source* (GNU LGPL) and is available from `http://jason.sourceforge.net`

[5] The '**~**' operator denotes strong negation in ***Jason***.

to the change in the mental attitudes of AgentSpeak agents when receiving messages from other agents (using a speech-act based language). This includes not only changes in beliefs and goals, but also the plans used by the agent. This allows agents to exchange know-how with other agents in the form of plans for dealing with specific events [3]. The intuitive idea is that if one does not know how to do something, one should ask someone who does. However, to systematise this idea, hence introducing the possibility of *cooperation* among agents, it was necessary not only the means for the retrieval of external plans for a given triggering event for which the agent has no applicable plan, but also to annotate plans with *access specifiers* (e.g., to prevent private plans being accessed by other agents), or with indications of what the agent should do with the retrieved plan once it has been used for a particular event (e.g., discard it, or keep it in the plan library for future reference).

Ontological reasoning In [17], an extension of AgentSpeak was proposed which aimed at incorporating ontological reasoning within an AgentSpeak interpreter. The language was extended so that the belief base can include Description Logic [4] operators; the extended language was called AgentSpeak-DL. In addition to the usual ABox (factual knowledge in the form of ground atomic formulæ), the belief base can also have a TBox (containing definitions of complex concepts and relationships between them). This results in a number of changes in the interpretation of AgentSpeak programs: (i) queries to the belief base are more expressive as their results do not depend only on explicit knowledge but can also be inferred from the ontology; (ii) the notion of belief update is refined so that a property about an individual can only be added if the resulting belief base is consistent with the concept description; (iii) the search for a plan (in the agent's plan library) that is relevant for dealing with a particular event is more flexible as this is not based solely on unification, but also on the subsumption relation between concepts; and (iv) agents may share knowledge by using web ontology languages such as OWL.

The issue of belief revision is clearly important in the context of ontological reasoning (e.g., item (ii) above), and this is another motivation for the work presented here. Further, ontologies are presently being used in various agent-based applications (see, e.g., [9]).

Although AgentSpeak-DL is not yet available in the latest release of ***Jason***, we briefly outline how our work on belief revision will combine with the ongoing implementation of AgentSpeak-DL. In ***Jason***, the abstract language presented in [17] will take the following more practical form. We will represent ontological knowledge in OWL Lite$^-$ [11], or in the form of Horn clauses. Interestingly, the OWL Lite$^-$ language was created precisely so that any ontology thus defined could be translated into Datalog, hence efficient query answering could be done based on logic programming techniques. Unlike the abstract language used in [17], definitions such as

$$presenter \equiv invitedSpeaker \sqcup paperPresenter.$$

are not allowed in the practical language to be used in this work (the best that we can do here are definitions such as $invitedSpeaker \sqsubseteq presenter$ and $paperPresenter \sqsubseteq presenter$). On the other hand, we will be able to express *ontology rules* [14] which are not expressible in description logic.

Belief annotations Another important change in the version of AgentSpeak interpreted by ***Jason*** is that atomic formulæ now can have "annotations". An annotation is a list of terms enclosed in square brackets immediately following a predicate. For example, the annotated belief "`green_patch(r1)[doc(0.9)]`" could be used by a programmer to represent the fact that rock `r1` is believed to have a green patch in it, and this is believed with a degree of certainty (`doc`) of 0.9. Within the belief base, an important use of annotations is to record the sources of information for a particular belief, and a (pre-defined) term `source(`s`)` is provided for that purpose, where s can be an agent's name (to denote the agent that has communicated that information), or two special atoms, `percept` and `self`, which denote, respectively, that a belief arose from perception of the environment, or from the agent explicitly adding a belief to its own belief base as a result of executing a plan. The initial beliefs that are part of the source code of an AgentSpeak agent are assumed to be internal beliefs (i.e., as if they had a `[source(self)]` annotation), unless the belief has any source explicit annotation given by the user (this could be useful if the programmer wants the agent to have an initial belief as if it had been perceived from the environment, or as if it had been communicated by another agent). For more on the annotation of sources of information for beliefs, see [16].

As will be seen below, annotations can be used to support context sensitive belief revision, where beliefs of a particular type or from a particular source are preferred to others when an inconsistency arises.

3.2 Belief Update in *Jason*

Users can customise certain aspects of the (practical) reasoning of a ***Jason*** agent by overriding methods of the Agent. This includes, for example, the three user-defined selection functions that are required by an AgentSpeak interpreter. One of the methods of the Agent class that can be overridden, which is of interest here, is the brf() method. This represents the *belief revision function* commonly found in agent architectures (although the Agents literature often assumes that this function is used mainly for belief update, rather than revision). To create a customised agent class which overrides the brf method (e.g., to include a more sophisticated algorithm than the standard one distributed with ***Jason***), the following method needs to be overridden[6]:

```
public class MyAgent extends Agent {

  public List[] brf(List adds, List dels) {
    // This function should revise the belief base
    // with the given literals to add and delete

    // In its return, List[0] has the list of actual
    // additions to the belief base, and List[1] has
    // the list of actual deletions; this is used to
    // generate the appropriate internal events
  }
}
```

[6] Note that the signature of the brf method as given below is different from what is currently available in ***Jason***, but this is how it will be in the next public release.

In the current ***Jason*** implementation, the brf method receives only a list of additions, and is used both for belief revision and belief update (i.e., perception of the environment is followed by a call to this method with literals representing the percepts[7]). For belief update following perception of the environment, it is assumed that all perceptible properties are included in the list of additions: all current beliefs no longer within the list of percepts are deleted from the belief base, and all percepts not currently in the belief base are added to it. For belief revision, the default brf method in ***Jason*** simply adds to the belief base any belief addition executed within a plan, as well as any information from trusted sources (note, however, that the source is annotated on the belief added to belief base, so in practice further consideration of the degree of trust in any belief can be taken by the programmer).

At present, belief additions (from whatever source) are *not* checked for consistency, with the result that the belief base can become inconsistent, unless much care is taken by programmers.

4 Requirements for Belief Revision in AgentSpeak

We have two main objectives in our introduction of belief revision in AgentSpeak. First the algorithm should be theoretically well motived, in the sense of producing revisions which conform to a generally accepted set of postulates characterising *rational* belief revision. Second, we want the resulting language to be practical, which means that the belief revision algorithm must be efficient. Our approach draws on recent work [2] on efficient (polynomial-time) belief revision algorithms which satisfy the well-known AGM postulates [1] characterising rational belief revision and contraction.

The theory of belief revision as developed by Alchourron, Gärdenfors, and Makinson in [12, 1, 13] models belief change of an idealised rational reasoner. The reasoner's beliefs are represented by a potentially infinite set of beliefs closed under logical consequence. When new information becomes available, the reasoner must modify its belief set to incorporate it. The AGM theory defines three operators on belief sets: expansion, contraction, and revision. *Expansion*, denoted $K + A$, simply adds a new belief A to K and the resulting set is closed under logical consequence. *Contraction*, denoted by $K \dot{-} A$, removes a belief A from from the belief set and modifies K so that it no longer entails A. *Revision*, denoted $K \dot{+} A$, is the same as expansion if A is consistent with the current belief set, otherwise it minimally modifies K to make it consistent with A, before adding A.

Contraction and revision cannot be defined uniquely, since in general there is no unique maximal set $K' \subset K$ which does not imply A. Instead, the set of 'rational' contraction and revision operators is characterised by the AGM postulates [1]. Below, $Cn(K)$ denotes closure of K under logical consequence.

The basic AGM postulates for contraction are:

(K$\dot{-}$1) $K \dot{-} A = Cn(K \dot{-} A)$ (closure)
(K$\dot{-}$2) $K \dot{-} A \subseteq K$ (inclusion)

[7] The fact that a literal is a percept rather than other forms of information is explicitly stated in the annotations: all percepts have a `source(percept)` annotation.

(K$\dot{-}$3) If $A \notin K$, then $K \dot{-} A = K$ (vacuity)

(K$\dot{-}$4) If not $\vdash A$, then $A \notin K \dot{-} A$ (success)

(K$\dot{-}$5) If $A \in K$, then $K \subseteq (K \dot{-} A) + A$ (recovery)

(K$\dot{-}$6) If $Cn(A) = Cn(B)$, then $K \dot{-} A = K \dot{-} B$ (equivalence)

AGM style belief revision is sometimes referred to as *coherence* approach to belief revision, because it is based on the ideas of coherence and informational economy. It requires that the changes to the agent's belief state caused by a revision be as small as possible. In particular, if the agent has to give up a belief in A, it does not have to give up believing in things for which A was the sole justification, so long as they are consistent with the remaining beliefs.

AGM belief revision is generally considered to apply only to idealised agents, because of the assumption that the set of beliefs is closed under logical consequence. To model AI agents, an approach called belief base revision has been proposed (see for example [15, 18, 24, 22]). A belief base is a finite representation of a belief set. Revision and contraction operations can be defined on belief bases instead of on logically closed belief sets. However the complexity of these operations ranges from NP-complete (full meet revision) to low in the polynomial hierarchy (computable using a polynomial number of calls to an NP oracle which checks satisfiability of a set of formulas) [20]. The reason for the high complexity is the need to check for classical consistency while performing the operations. One way around this is to weaken the language and the logic of the agent so that the consistency check is no longer an expensive operation (as suggested in [19]). This is also the approach taken in [2] and adopted here.

The 'language' of an AgentSpeak agent is weaker than the language of full classical logic (the belief base contains only literals) and the deductions the agent can make are limited to what can be expressed as plans (and, for example, ontology rules). We introduce belief revision operators in AgentSpeak which satisfy all but one of the AGM postulates (recovery is not satisfied), but the logical closure Cn in the postulates is interpreted as closure with respect to a logic which is weaker than full classical logic. This allows us to define theoretically sound, but efficient belief revision operations.

Another strand of theoretical work in belief revision is the *foundational*, or *reason-maintenance* style approach to belief revision. Reason-maintenance style belief revision is concerned with tracking dependencies between beliefs. Each belief has a set of justifications, and the reasons for holding a belief can be traced back through these justifications to a set of foundational beliefs. When a belief must be given up, sufficient foundational beliefs have to be withdrawn to render the belief underivable. Moreover, if all the justifications for a belief are withdrawn, then that belief itself should no longer be held. Most implementations of reason-maintenance style belief revision are incomplete in the logical sense, but tractable.

In the next section we present an approach to belief revision and contraction for resource-bounded agents which allows both AGM and reason-maintenance style belief revision.

5 The Belief Revision Algorithm

In this section we briefly describe the linear-time contraction algorithm introduced in [2]. The algorithm defines resource-bounded contraction by a literal A as the removal of A and sufficient literals from the agent's belief base so that A is no longer derivable.

Assume that the agent's belief base is a directed graph, where the nodes are beliefs and *justifications*. A justification consists of a belief and a *support list* containing the context (and possibly the triggering event) of the plan used to derive this belief, for example: $(A, [B, C])$, where A is a derived belief and it was asserted by a plan with context B and triggering belief addition C (or derived by an ontology rule $B, C \rightarrow A$). If A can be derived in several different ways, for example, from B, C and from D (where B, C and D are in the belief base), the graph contains several justifications for A, for example $(A, [B, C])$ and $(A, [D])$. Foundational beliefs which were not derived, have a justification of the form $(D, [])$. In the graph, each justification has one outgoing edge to the belief it is a justification for, and an incoming edge from each belief in its support list. We assume that each support list s has a designated *least preferred* member $w(s)$. Intuitively, this is a belief which is not preferred to any other belief in the support list, and which we would be prepared to discard first, if we have to give up one of the beliefs in the list. We discuss possible preference orderings and their computation in the next section. We assume that we have constant time access to $w(s)$.

The algorithm to contract a belief A is as follows:

```
For each of A’s outgoing edges
    to a justification (C, s),
    remove (C,s) from the graph.

For each of A’s incoming edges
    from a justification (A, s),
        if s is empty:
            remove (A, s);
        else:
            contract by w(s);
Remove A.
```

To implement reason-maintenance type contraction, we also remove beliefs which have no incoming edges.

In [2], it was shown that the contraction operator defined by the algorithm satisfies (K$\dot{-}$1)–(K$\dot{-}$4) and (K$\dot{-}$6). The agent's beliefs are closed under logical consequence in in a logic W which has a single inference rule (generalised modus ponens):

$$\frac{\delta(A_1), \ldots, \delta(A_n), \qquad \forall \bar{x}(A_1 \wedge \ldots \wedge A_n \rightarrow B)}{\delta(B)}$$

where δ is a substitution function which replaces all free variables of a formula with constants.

The algorithm runs in time $O(kr + n)$, where k the maximal number of beliefs in any support list, r is the number of plans, and n the number of literals in the belief base [2].

5.1 Preferred Contractions

In general, an agent will prefer some contractions to others. In this section we focus on contractions based on preference orders over individual beliefs, e.g., degree of belief or commitment to beliefs.

We distinguish *independent* beliefs, beliefs which have at least one non-inferential justification (i.e., a justification with an empty support), such as beliefs acquired by perception and the literals in the belief base when the agent starts. We assume that an agent associates an *a priori* quality with each non-inferential justification for its independent beliefs. For example, communicated information may be assigned a degree of reliability by its recipient which depends on the degree of reliability of the speaker (i.e., the speaker's reputation), percepts may be assumed to be more reliable than communicated information, and so on.

For simplicity, we assume that quality of a justification is represented by non-negative integers in the range $0, \ldots, m$, where m is the maximum size of the belief base. A value of 0 means the lowest quality and m means highest quality. We take the preference of a literal A, $p(A)$, to be that of its highest quality justification:

$$p(A) = max\{qual(j_0), \ldots, qual(j_n)\},$$

where $j_0, \ldots, j_n$ are all the justifications for A, and define the quality of an inferential justification to be that of the least preferred belief in its support: [8]

$$qual(j) = min\{p(A) : A \in \text{ support of } j\}.$$

This is similar to ideas in argumentation theory: an argument is only as good as its weakest link, yet a conclusion is at least as good as the best argument for it. This approach is also related to Williams 'partial entrenchment ranking' [25] which assumes that the entrenchment of any sentence is the maximal quality of a set of sentences implying it, where the quality of a set is equal to the minimal entrenchment of its members. While this approach is intuitively appealing, nothing hangs on it, in the sense that any preference order can be used to define a contraction operation, and the resulting operation will satisfy the postulates. To perform a preferred contraction, we preface the contraction algorithm given above with a step which computes the preference of each literal in the belief base, and for each justification, finds the position of a least preferred member of the support list. The preference computation algorithm can be found in [2].

We then simply run the contraction algorithm, to recursively delete the weakest member of each support in the dependencies graph of A.

We define the *worth* of a set of literals Γ as $worth(\Gamma) = max\{p(A) : A \in \Gamma\}$. In [2] it was shown that the contraction algorithm removes the set of literals with the least worth. More precisely:

Proposition 1. *If contraction of the set of literals in the belief base K by A resulted in removal of the set of literals Γ, then for any other set of literals Γ' such that $K - \Gamma'$ does not imply A, $worth(\Gamma) \leq worth(\Gamma')$.*

[8] Literals with no supports (as opposed to an empty support) are viewed as having an empty support of the lowest quality.

The proof is given in [2]. Computing preferred contractions involves only modest computational overhead. The total cost of computing the preference of all literals in the belief base is $O(n \log n + kr)$, where n the number of literals in the belief base, k is the maximal number of beliefs in any support list, and r the number of plans. As the contraction algorithm is unchanged, this is also the additional cost of computing a preferred contraction. Computing the most preferred contraction can therefore be performed in time linear in $kr + n$.

5.2 Revision

In the previous sections we described how to contract by a belief. Now let us consider revision, which is adding a new belief in a manner which does not result in an inconsistent set of beliefs.

If the agent is a reasoner in classical logic, revision is definable in terms of contraction and vice versa using Levi identity $K \dot{+} \overset{df}{=} (K \dot{-} \neg A) + A$ and Harper identity $K \dot{-} A \overset{df}{=} (K \dot{+} \neg A) \cap K$ (see [13]).

However, revision and contraction are not inter-definable in this way for an agent which is not a classical reasoner, in particular, a reasoner in a logic for which it does not hold that $K + A$ is consistent if, and only if, $K \not\vdash \neg A$. If we apply the Levi identity to the contraction operation defined earlier, we will get a revision operation which does not satisfy the belief revision postulates. One of the reasons for this is that contracting the agent's belief set by $\neg A$ does not make this set consistent with A, so $(K \dot{-} \neg A) + A$ may be inconsistent.

Instead, we define revision of the set of literals in the belief base K by A as $(K + A) \dot{-} \bot$ (add A, close under consequence, and eliminate all contradictions).

```
Algorithm: revision by A

Add A to K;
apply all matching plans;

while there is a pair (B, ~B) in K:
   contract by the least preferred member of the pair
```

In [2], it is shown that this definition of revision satisfies all of the basic AGM postulates for revision below apart from (K$\dot{+}$2):

(K$\dot{+}$1) $K \dot{+} A = Cn(K \dot{+} A)$
(K$\dot{+}$2) $A \in K \dot{+} A$
(K$\dot{+}$3) $K \dot{+} A \subseteq K + A$
(K$\dot{+}$4) If $\{A\} \cup K$ is consistent, then $K + A = K \dot{+} A$[9]
(K$\dot{+}$5) $K \dot{+} A$ is inconsistent if, and only if, A is inconsistent.
(K$\dot{+}$6) If $Cn(A) = Cn(B)$, then $K \dot{+} A = K \dot{+} B$

[9] We replaced '$\neg A \notin K$' with '$\{A\} \cup K$ is consistent' here, since the two formulations are classically equivalent.

6 Belief Revision in *Jason*

Future releases of ***Jason*** will include an alternative definition of the brf() method discussed in Section 3.2 which implements the belief revision algorithm presented above. A belief to be added to the belief base, passed to this new implementation of brf, may be discarded or may result in the deletion of some other belief(s) in order to allow the new belief to be consistently added to the belief base. Which beliefs are effectively deleted is determined by a user-specified preference order (see below).

The only change to the AgentSpeak interpreter code that was necessary to facilitate the implementation of the belief revision algorithm, was to explicitly include in any internal belief change, the label of the plan that executed the belief change. For example, if at a particular reasoning cycle, the intended means (i.e., plan instance) chosen for execution is "`@p1` te `:` ct `<- +b.`", the belief b is annotated with "`plan(p1)`" (in addition to `source(self)`, as normally) before adding b to the belief base.

The graph used by the belief revision algorithm is implemented in terms of two lists for each belief: the "dependencies list" (the literals that allowed the derivation of the belief literal in question), and the "justifies list" (which other beliefs the literal in question justifies, i.e., it appears in their dependencies list).[10] Each belief to be added has an annotation "`plan()`" recording the label of the plan instance that generated it, which can be used to retrieve the necessary information regarding the antecedents of the belief from the plan library (together with the unifier used in that plan's instance in the set of intentions). For example, if the plan that generated the belief change, say $+bl$, has the form "`@p` te `:` l_1 `& ...&` l_n `<-` bd", where te is a triggering event and bd a plan body, the support list of the justification is simply the (ground) literals from the plan context, "[l_1,...,l_n]". Note that if the triggering event, te, is itself a belief (addition), the literal in te is included together with the context literals in the support list. Further, for each literal in $l_1, \ldots, l_n$ we add the justification to the literal's "justifies" list. We also record the time at which the justification was added to the relevant list.

In addition to the "dependencies" and "justifies" lists, the belief revision algorithm also requires the definition of a partial order relation specifying contraction preference. To allow for user customisation, this is defined as a separate method that can also be overridden. The default definition of this method gives preference to perceived information over communicated information (as also happens in [23]), and in case of information from similar sources, it gives preference to newer information over older information (this is why the time when a justification was inserted is also annotated, as explained above).

The implementation described above is conservative in revising only the agent's belief state. The agent's plans are considered part of the agent's program and are not revised (though revising, e.g., plans received from other agents would be an interesting extension). Similarly, when revising beliefs derived using ontological rules, we assume the ontology used by the agent to be immutable and consistent and that it is consis-

[10] Note that the "dependencies" and "justifies" lists are associated with each unique belief, i.e., a ground belief atom and the annotations with which it is asserted into the belief base, rather than the internal ***Jason*** representation of the belief which holds all annotations for a given ground belief atom in a single list.

tent with every other ontology it references. Moreover, *intention* revision remains the responsibility of the programmer. Changes in the agent's intentions following the removal of beliefs to restore consistency must be programmed using the appropriate ***Jason*** mechanisms. All belief changes, regardless of whether they are internal, communicated, or perceived can lead to the execution of a plan which could be used, for example, to drop an intention. If the belief revision algorithm has to remove any beliefs to ensure consistency, this will also generate the appropriate (belief-deletion) internal events, which in turn can trigger the execution of a such plans to revise the agent's intentions.

7 An Example

To illustrate the importance of belief revision in the context of AgentSpeak, we present a simple example of an agent that buys stocks from the stock market. The agent receives financial information (or guesses) from other agents, some of which can be trusted (or are currently considered trustworthy), and it also has access to Web Services which filter relevant newspaper stories and provide symbolic versions of such news for stock market agents. As these web services are authenticated, this corresponds to actual perception of the "environment".

Suppose our agent receives a message $\langle \mathtt{ag1}, tell, \mathtt{salesUp}(\mathtt{c1}) \rangle$ and its plan library has the following plan:

```
+salesUp(C)[source(A)]
  : wellManaged(C) & trust(A)
    <- +goodToBuy(C).
```

When the plan is executed, the brf() method will then add `goodToBuy(c1)[source(ag1)]` to the belief base with `[salesUp(c1), wellManaged(c1), trust(ag1)]` in its "dependencies" list, and `goodToBuy(c1)` is added to the "justifies" lists of the beliefs `salesUp(c1)`, `wellManaged(c1)`, and `trust(ag1)`. In the context of the overall agent program, the idea is that if the agent ever comes to have the goal of buying stocks, it can make use of beliefs such as `goodToBuy`, together with various other conditions, to decide which specific stocks to buy.

Now assume that from the financial news web service, the agent acquires the belief `stocks(c2,10)[source(percept)]`, which means that company `c2`'s stocks are up by 10 points, and the agent also believes that `rival(c2,c1)` (i.e., that companies `c2` and `c1` are competitors), so that increase in the stocks of one of them tend to lead to decrease in the other's stocks. Assume further that the agent happens to have the following plan:

```
+stocks(C,P)
  : P > 5 & rival(C,R)
    <- +~goodToBuy(R).
```

When the plan is executed, the attempt to simply add `~goodToBuy(c1)` to the belief base would not be carried out because it would result in an inconsistent belief state. With the available contraction preference relation, it is not difficult to see that in this instance, the algorithm would contract `goodToBuy(c1)` because its support is based on communicated information which is less reliable than the observed information from which `~goodToBuy(c1)` was derived.

As can be seen, the belief revision algorithm takes care of ensuring that inconsistencies such as (`goodToBuy(c1)` $\wedge$ `~goodToBuy(c1)`) never occur in the belief base. Moreover, the data structures used by the algorithm (the dependencies and justifications lists) allow it to automatically revise the belief base in ways that previously would require major programming efforts from the user. For example, suppose the agent receives news that a crooked CEO has just been fired from `c1`. The agent is likely to have a plan to update its beliefs about `c1` being well managed as a consequence of such new information about the CEO. If the user has chosen the *reason-maintenance style* of the algorithm, and there is no other justification for `goodToBuy`, then the algorithm would remove not only the `wellManaged(c1)` belief, but also the `goodToBuy(c1)` belief because the latter depends on the former. Similarly if for some reason the agent later finds out that `ag1` is not trustworthy after all.

However, with the user choosing the *coherence style* of the algorithm, removing `wellManaged` would *not* remove `goodToBuy`. Although in this example the reason-maintenance style is clearly more adequate, in other applications the coherence style might be more useful. In any case, it is clear that without the use of an automatic belief revision algorithm, it would be very difficult for a programmer to ensure such kind of revision would occur appropriately at all times. This would require that the programmer developed an application-specific brf method, or else writing specific plans to handle all possible events (due to belief changes) that might affect any such inferences.

8 Conclusions and Future Work

As multi-agent programming languages become richer, it becomes harder for programmers to ensure that the belief states of agents developed using these languages are kept consistent. In this paper we briefly summarised the rationale for including automatic belief revision in an agent programming language. Using the AgentSpeak programming language as an example, we showed how a number of features recently added to the language dramatically increase the need for automatic belief revision. We motived the choice of a polynomial-time belief revision algorithm and described its integration into the ***Jason*** AgentSpeak interpreter. We also gave a simple example which illustrates the utility of such an automatic belief revision mechanism in a practical multi-agent system application, and sketched how it can significantly reduce the programming efforts required. We believe that other agent-oriented programming languages and their platforms [5], which currently push responsibility for maintaining a consistent belief state onto programmers, can also benefit from our approach.

We are aware of a number of limitations of the work presented here. In future work, we plan to further explore the issue of the interaction of belief revision with the ***Jason*** extension that allows the belief base to refer to OWL ontologies and uses ontological reasoning as part of the AgentSpeak interpreter [17], and to address the issue the inferences that occur from complex test goals. On the more practical side, we plan to develop large-scale agent applications to assess the performance of ***Jason*** with belief revision.

Acknowledgements

Rafael Bordini gratefully acknowledges the support of The Nuffield Foundation (grant number NAL/01065/G).

References

1. C. E. Alchourrón, P. Gärdenfors, and D. Makinson. On the logic of theory change: Partial meet functions for contraction and revision. *Journal of Symbolic Logic*, 50:510–530, 1985.
2. N. Alechina, M. Jago, and B. Logan. Resource-bounded belief revision and contraction. In *Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT 2005)*, Utrecht, the Netherlands, July 2005.
3. D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19–23 July*, pages 698–705, New York, NY, 2004. ACM Press.
4. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *Handbook of Description Logics*. Cambridge University Press, Cambridge, 2003.
5. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.
6. R. H. Bordini, J. F. Hübner, et al. ***Jason**: A Java-based agentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, release version 0.7 edition, August 2005. `http://jason.sourceforge.net/`.
7. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini et al. [5], chapter 1.
8. R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226, Sept. 2004. Special Issue on Computational Logic in Multi-Agent Systems.
9. H. Chen, T. Finin, and A. Joshi. The SOUPA Ontology for Pervasive Computing. In V. T. et al, editor, *Ontologies for Agents: Theory and Experiences*, pages 233–258. BirkHauser, 2005.
10. K. L. Clark and F. G. McCabe. Ontology schema for an agent belief store. *IJCIS*, 2006. To appear.
11. J. de Bruijn, A. Polleres, and D. Fensel. Owl lite^{-}. working draft, wsml delieverable d20 v0.1, WSML, 18th July 2004. `http://www.wsmo.org/2004/d20/v0.1/20040629/`.
12. P. Gärdenfors. Conditionals and changes of belief. In I. Niiniluoto and R. Tuomela, editors, *The Logic and Epistemology of Scientific Change*, pages 381–404. North Holland, 1978.
13. P. Gärdenfors. *Knowledge in Flux: Modelling the Dynamics of Epistemic States*. The MIT Press, Cambridge, Mass., 1988.
14. I. Horrocks and P. F. Patel-Schneider. A proposal for an OWL rules language. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004*, pages 723–731. ACM, 2004.
15. D. Makinson. How to give it up: A survey of some formal aspects of the logic of theory change. *Synthese*, 62:347–363, 1985.

16. Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, Proc. of the First Int. Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*, number 2990 in LNAI, pages 135–154, Berlin, 2004. Springer-Verlag.
17. A. F. Moreira, R. Vieira, R. H. Bordini, and J. Hübner. Agent-oriented programming with underlying ontological reasoning. In *Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT 2005)*, Utrecht, the Netherlands, July 2005.
18. B. Nebel. A knowledge level analysis of belief revision. In R. Brachman, H. J. Levesque, and R. Reiter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the First International Conference*, pages 301–311, San Mateo, 1989. Morgan Kaufmann.
19. B. Nebel. Syntax-based approaches to belief revision. In P. Gärdenfors, editor, *Belief Revision*, volume 29, pages 52–88. Cambridge University Press, Cambridge, UK, 1992.
20. B. Nebel. Base revision operations and schemes: Representation, semantics and complexity. In A. G. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 341–345, Amsterdam, The Netherlands, August 1994. John Wiley and Sons.
21. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proceedings of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in Lecture Notes in Artificial Intelligence, pages 42–55, London, 1996. Springer-Verlag.
22. H. Rott. "Just Because": Taking belief bases seriously. In S. R. Buss, P. Hájaek, and P. Pudlák, editors, *Logic Colloquium '98—Proceedings of the 1998 ASL European Summer Meeting*, volume 13 of *Lecture Notes in Logic*, pages 387–408. Association for Symbolic Logic, 1998.
23. R. M. van Eijk, F. S. de Boer, W. van der Hoek, and J.-J. C. Meyer. Information-passing and belief revision in multi-agent systems. In J. P. Müller, M. P. Singh, and A. S. Rao, editors, *Intelligent Agents V — Agent Theories, Architectures, and Languages, 5th International Workshop, ATAL '98, Paris, France, July 4-7, 1998, Proceedings*, volume 1555 of *LNCS*, pages 29–45, Berlin, 1999. Springer-Verlag.
24. M.-A. Williams. Two operators for theory base change. In *Proceedings of the Fifth Australian Joint Conference on Artificial Intelligence*, pages 259–265. World Scientific, 1992.
25. M.-A. Williams. Iterated theory base change: A computational model. In *Proceedings of Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1541–1549, San Mateo, 1995. Morgan Kaufmann.
26. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.

A Foundational Ontology of Organizations and Roles

Guido Boella[1] and Leendert van der Torre[2]

[1]Dipartimento di Informatica - Università di Torino - Italy. email: guido@di.unito.it
[2]University of Luxembourg. e-mail: leendert@vandertorre.com

Abstract. In this paper we propose a foundational ontology of the social concepts of organization and role which structure institutions. We identify which axioms model social concepts like organization and role and which properties distinguish them from other categories like objects and agents: the organizational structure of institutions, the relation between roles and organizations, and the powers among the components of an organization. All social concepts depend on descriptions defining them, which are collectively accepted, and the description defining the components of organizations, including roles, are included in the description of the organizations they belong to. Thus, the relational dependence of roles means that they are defined in the organizations they belong to. Finally, powers inside organizations are defined by the fact that components of an organization can access the state of the organization whose definition they depend on and of the other components, thus violating the standard encapsulation principle of objects.

1 Introduction

In order to constrain the autonomy of agents and to control their emergent behavior in multiagent systems, the notion of organization has been applied [1]. According to Zambonelli *et al.* [2] "a multiagent system can be conceived in terms of an organized society of individuals in which each agent plays specific roles and interacts with other agents". For Zambonelli *et al.* "an organization is more than simply a collection of roles [...] further organization-oriented abstractions need to be devised and placed in the context of a methodology [...] As soon as the complexity increases, modularity and encapsulation principles suggest dividing the system into different sub-organizations".

There is not yet a common agreement, however, on how to model organizations and roles, and, in particular, which are the ontological assumptions behind them. For example, departments and roles are parts of an organization, but they do not exist without it. Can organizations be explained by means of agent based models? Or can they be better modelled with the object oriented paradigm?

Since the existence of institutions depends on what Searle [3] calls the construction of social reality, it is possible that institutions, organizations and roles have very different properties with respect to objects or agents. Searle argues that social reality is constructed by means of so called "constitutive rules" which state what "counts as" institutional facts in the institution. Constitutive rules define institutions: they exist only because of the collective acceptance of constitutive rules by a community.

Searle's construction of social reality does not explain all issues, in particular, the fact that some institutions have a structure in terms of sub-institutions and roles. We will call them organizations. Thus Searle's analysis is not a sufficient starting point for a foundational ontology, that specifies which are the properties distinguishing social reality from objects and agents. We need to know the axioms which allow to distinguish them from, rather than specifying all the properties of organizations, including those in common with agents. Thus the research questions of this paper are:

– How do organizations and roles differ from objects and agents?
– How can a foundational ontology of social entities, like organizations and roles, be constructed?

In [4–6] we start studying some properties of social entities. However, these works are based on a very specific multiagent framework, which uses the so called agent metaphor, i.e., the attribution of mental attitudes to social entities to explain them.

So in this paper we analyse organizations using an axiomatic ontology and we consider additional properties. The methodology we choose is to extend the ontology of Masolo *et al.* [7]. The main properties of their framework are three. First, it allows to express the fact that social concepts are defined by means of descriptions. Second, it explains the definitional dependence of a role from another concept and the relational nature of roles. Last, it offers a temporalized classification relation, used for modelling the fact that roles are anti-rigid.

We extend Masolo *et al.* [7]'s axiomatic ontology to model institutions and their organizational structure, to explain the asymmetry in the relations defining roles, and to introduce the notion of power relations internal to the organizations. With this work we want to justify the decisions taken in our other works about normative systems, organizations and roles, showing that they all share a common denominator. Second, we want to show that current object oriented representation languages like UML can be extended using the ontology developed in this work, so to ensure a large applicability.

This paper is structured as follows. First, we consider the differences between social reality and objects and agents. In Section 3, we present Masolo *et al.* [7]'s model. In Section 4, starting from the limitations of [7] we extend it to define the foundational ontology. In Section 5, we consider the relation of this ontology with our other works. Conclusions end the paper.

2 The properties of organizations

The role of knowledge representation and software engineering is to provide models and techniques that make it easier to handle the complexity arising from the large number of interactions in a system [8]. Models and techniques allow expressing knowledge and supporting the analysis and reasoning about a system to be developed. As the context and needs of software change, advances are needed to respond to changes. For example, today's systems and their environments are more varied and dynamic, and accommodate more local freedom and initiative [9].

For these reasons, agent orientation emerged as a new paradigm for designing and constructing software systems [8, 9]. The agent oriented approach advocates decomposing problems in terms of autonomous agents that can engage in flexible, high-level interactions. Much like the concepts of activity and object that have played pivotal roles in earlier modelling paradigms - Yu [9] argues - the agent concept can be instrumental in bringing about a shift to a much richer, *socially-oriented ontology* that is needed to characterize and analyze today's systems and environments.

The notions of institution, organization and role are part of this socially-oriented ontology. It is not clear, however, if the ontological assumptions behind this kind of entities are the same which underlie objects and agents. Many approaches recognize as properties of social entities their being the addressee of obligations [10], like agents are, the delegation mechanisms among roles [11], *etc*. Organizations are modelled as collections of agents, gathered in groups [1], playing roles [8, 12] or regulated by organizational rules [2]. We focus instead on the distinguishing properties of social concepts of organization and role.

Consider, for example, an organization which is composed by a direction area and a production area. The direction area is composed by the CEO and the board. The board is composed by a set of administrators. The production area is composed by two production units; each production unit by a set of workers. The direction area, the board, the production area and the production units are *sub-organizations*. In particular, the direction area and the production areas belong to the organization, the board to the direction area, *etc*. The CEO, the administrators and the members of the production units are *roles*, each one belonging to a sub-organization, e.g., the CEO is part of the direction area. This recursive decomposition terminates with roles: roles, unlike organizations and sub-organizations, are not composed by further social entities. Rather, roles are played by other agents, real agents who have to act as expected by their role.

Besides the decomposition structure, as we argue in [6] in organizations we have relations among the components of the organization which specify which are the powers of each component to modify the institutional properties of the other component institutions. This relation does not necessarily matches the decomposition hierarchy. For example, the senior board member has the power to command other members of the board to participate to a board meeting, even if it is at the same decomposition level of the other members. Moreover, the head of a department can give commands to the other members of the department even if they are roles all at the same level. Viceversa, the CEO and the board can take decisions for the whole organization they belong to, for example, committing it to pay for a purchased good.

Is it possible to model such structures in the object oriented paradigm? The object oriented paradigm is based on the idea that software design and implementation can be inspired by our commonsense view of the reality made of objects. For Booch [13] a basic property of objects is that they can be decomposed. Decomposition allows coping with complexity: "the most basic technique for tackling any large problem is to divide it into smaller, more manageable chunks each of which can then be dealt with in relative isolation". Isolation is the idea that code should be encapsulated in classes hiding the implementation of the objects' state; thus, other objects can access an object's state only via its public interface. Decomposition means that an object can include other objects which exist independently of it, like they were parts of the object. But even the components of an object can access it only via its public interface (and vice-versa) to preserve the encapsulation principle.

In case of organizations, the situation is different. First, in the decomposition structure: the components of an organization do not exist independently from the organization itself. For example a department does not exist without the organization it belongs to. If an organization goes bankrupt its departments do not exist anymore and similarly the roles in them (there is no CEO nor employee anymore). Viceversa, an organization can close a department without necessarily giving up its identity. Second, the notion of power inside an organization conflicts with the encapsulation principle of objects.

One alternative could be to see whether organizations can be modelled as agents, but again some difficulties arise. First of all, organizations can have organizations as their parts, while it is debatable whether agents can have parts which are homogeneous with the whole. Moreover, agents can play roles but they cannot have roles as their parts.

However, some form of decomposition should be added to multiagent systems, as noticed by Zambonelli *et al.* [2]: agents alone, and also roles, are not sufficient to deal with the complexity of a system; an organizational structure added to a multiagent system fosters modularity and encapsulation.

A bigger problem is that while agents are autonomous, organizations and roles are not, in two senses. First of all, roles' decisions are taken by the players of the roles: the actions of their players count as decisions of the roles. Analogously, the organizations take a decision on the basis of the decisions of their roles (e.g., the CEO) or sub-organizations (e.g., the board). These relations among decisions are expressed via constitutive rules. Second, a role is not autonomous in the sense that it cannot decide which goals to adopt. Rather, the goals representing the responsibilities of a role are delegated to it by other roles or by the organization itself. For example, an employee can be commanded to perform a task by its director. Analogously, the role's beliefs are assigned to it by other roles and organizations: consider the case of an advocate in a trial who has to show to believe and to support the belief that his client is innocent even if he privately believes otherwise.

We do not consider in this paper, instead, the control structure of organizations, which, e.g., [11] discuss.

Again these relations are expressed via constitutive rules, saying, e.g., that a decision of another role counts as the adoption of a goal by a role. Constitutive rules have been introduced by Searle in its construction of social reality:

> "Some rules regulate antecedently existing forms of behaviour. For example, the rules of polite table behaviour regulate eating, but eating exists independently of these rules. Some rules, on the other hand, do not merely regulate an antecedently existing activity called playing chess; they, as it were, create the possibility of or define that activity. The activity of playing chess is constituted by action in accordance with these rules. The institutions of marriage, money, and promising are like the institutions of baseball and chess in that they are systems of such constitutive rules or conventions" ([14], p. 131).

For Searle, regulative and constitutive norms are related via institutional facts like marriage, money and private property. They emerge from an independent ontology of "brute" physical facts through constitutive rules of the form "such and such an X counts as Y in context C" where X is any object satisfying certain conditions and Y is a label that qualifies X as being something of an entirely new sort. E.g., “X counts as a presiding official in a wedding ceremony”, “this bit of paper counts as a five euro bill” and “this piece of land counts as somebody’s private property”.

As we say in [6, 15] constitutive rules define the powers among roles and organizations. Powers are behaviors which affect the internal state of another entity (the decision and obligations of an organization, the goals and beliefs of a role, *etc.*) [6]. In particular, in the example above we can distinguish three kinds of power of roles, which we extend here also to sub-organizations:

- Actions of a sub-organization or of a role that are recognized as actions of the organization: e.g., a CEO’s signature on a buy-order, or a decision of the board, is considered as a commitment of its organization to pay for the requested good.
- Actions of the agent playing the role that can modify the state of the role itself. E.g., a director can commit itself to new responsibilities.
- Interaction capabilities among sub-organizations and roles in the same organization. The CEO or the board can send a message to another role, e.g., a command to an employee.

Powers do not only violate the autonomy of organizations, but they violate also the standard encapsulation principle in object orientation described above: a sub-organization or a role which are part of an organization can access the private state of the organization they belong to and of other roles and vice versa (but not the state of the agents playing them, which are autonomous).

If we consider the current ontological analyses of social reality, we find that further differences between organizations and objects and agents have been identified. When roles are considered as predicates like natural kinds (from the linguistic analogy between “John is a person” and “John is a student”), as e.g. [7] do, then there is an asymmetry: John can stop being a student, but he cannot stop being a person. A role like student is anti-rigid because persons are only contingently students. This is a problem for the notion of class used in agent and object orientation which lacks of dynamic reclassification.

Furthermore, roles and sub-organizations are defined in relation to the organizations they belong too. In contrast, the other kinds of entities are defined independently of one another’s definition (albeit in their definitions other concepts are used). This is called *definitional dependence*. This property cannot be accounted for by the current view of object orientation and agent orientation.

Finally we will not consider here the problem of collective acceptance of institutions: institutions do not exist by themselves but they exist only if their definitions in terms of constitutive rules are collectively accepted by the community of agents.

3 Background

Masolo *et al.* [7] present a formal framework for developing axiomatical ontologies of socially constructed entities, and study the ontological nature of roles. Social entities and roles exist just because of social conventions, i.e., constitutive rules accepted by communities of agents: these can be social concepts like organization, nation, money, or social individuals like the DALT workshop or the FIAT company.

In Masolo *et al.* [7] roles are 'properties' according the position defended by Sowa [16]: roles can be 'predicated' of different entities, i.e., different entities can play the same role. The basic properties of roles are the anti-rigidity and being founded. According to Guarino and Welty [17] the definition of foundation is: "a property a is founded on a property b if, necessarily, for every instance x of a there exists an instance y of b which is not 'internal' to x". The notion of 'internalness' is complex: e.g., if x is a car, things internal to it can be parts of it (its wheels), but also constituents of it (the metal it is made of) or qualities of it (its color). To avoid all trivial cases, Fine [18] introduces another notion of dependence: "to say that an object x depends upon an F is to say that an F will be ineliminably involved in any definition of x".

This notion can be generalized to properties considering that a property a is *definitionally dependent* on a property b if, necessarily, any *definition* of a ineliminably involves b. To model this fact 'definitions' are explicitly introduced in the domain of discourse. [7] consider 'reified' social concepts and roles, as well as their descriptions, i.e, the 'social conventions' that define them. This allows to formally characterize in a first-order theory the relationships among all these entities and to talk of roles as 'first-class citizens', similarly to more common entities like objects, events, *etc.*

[7]'s approach is based on a distinction between the properties and relations in the ground ontology (like DOLCE [19]) and those at the object level representing the social reality. The former ones are represented as predicates and therefore assumed as static, rigid, extensional, and not explicitly defined or linked to a description (i.e., the primitive predicates of the theory). The latter ones (called "concepts") are reified and not necessarily static, rigid, and extensional and for which it is possible to explicitly describe some aspects of the conventions that define them (called "descriptions").

Social concepts, denoted by $CN(x)$ are defined (DF) or used (US) by descriptions (DS) and they classify (CF) other individuals: $DF(x, y)$ stands for "the concept x is defined by the description y" to deal with the social, relational, and contextual nature of social concepts. $US(x, y)$ stands for "the concept x is used by the description y"; they introduce a temporalized classification relation to link concepts with the entities they classify, while accounting for the dynamic behavior of social roles: $CF(x, y, t)$ stands for "at the time t, x is classified by the concept y" or, more explicitly, "at the time t, x satisfies all the constraints stated in the description of y".

In the axioms defining [7]'s theory, $ED(x)$ stands for "x is an endurant", i.e., an entity that is wholly present at any time it is present, e.g., a book, Hakodate, a law, some metal, *etc.* $NASO(x)$ stands for "x is a non-agentive social object", i.e., an endurant that: (i) is not directly located in space and, has no direct spatial qualities; (ii) has no intentionality; (iii) depends on a community of intentional agents, e.g., a law, an organization, a currency, an asset *etc.*; $TL(x)$ stands for "x is a temporal location", i.e., a temporal interval or instant; $P(x, y)$ stands for "x is part-of y", for perdurants and temporal locations; $PRE(x, t)$ stands for "x is present at the time t".

We report here the most important axioms of their theory. Concepts, and descriptions as well, are non-agentive social objects; concepts are linked to descriptions by the relations used-by (US) and defined-by (DF). Theorem T2 below captures the fact that a concept must be defined by a single description. This is not true for the US relation: concepts can be used by different descriptions.

(A1) $DS(x) \supset NASO(x)$
(A2) $CN(x) \supset NASO(x)$
(A3) $DS(x) \supset \neg CN(x)$
(A4) $US(x, y) \supset (CN(x) \wedge DS(y))$
(A5) $DF(x, y) \supset US(x, y)$
(A8) $(DF(x, y) \wedge DF(x, z)) \supset y = z$
(T1) $DF(x, y) \supset (CN(x) \wedge DS(y))$
(T2) $CN(x) \supset \exists! y(DF(x, y))$
(A11) $CF(x, y, t) \supset (ED(x) \wedge CN(y) \wedge TL(t))$
(A14) $CF(x, y, t) \supset \neg CF(y, x, t)$
(A15) $(CF(x, y, t) \wedge CF(y, z, t)) \supset \neg CF(x, z, t)$

The properties of anti-rigidity (AR) and foundation (FD) for roles can be defined in this formalism. A concept is anti-rigid if, for any time an entity is classified under it, there exists a time at which the entity is present but not classified under the concept:

(D1) $AR(x) \equiv_{df} \forall y, t(CF(y, x, t) \supset \exists t'(PRE(y, t') \wedge \neg CF(y, x, t')))$

A concept x is founded if its definition involves (at least) another concept y (definitional dependence) such that for each entity classified by x, there is an external entity classified by y:

(D2) $FD(x) \equiv_{df} \exists y, d(DF(x, d) \wedge US(y, d) \wedge$
$\forall z, t(CF(z, x, t) \supset \exists z'(CF(z', y, t) \wedge \neg P(z, z', t) \wedge \neg P(z', z, t)))$

Roles are anti-rigid and founded:

(D3) $RL(x) \equiv_{df} AR(x) \wedge FD(x)$

Masolo *et al.* [20] extend [7]'s framework introducing explicitly a relation between an institution and a role to express that a role like student is relationally dependent, e.g., for a person to be a student it requires the existence of another entity, namely a certain university, to which this person is related by an enrollment relation. As Steimann [21] shows, this view of roles as anti-rigid and relationally dependent predicates is supported by the vast majority of approaches in the conceptual modeling and object-modeling literature.

Roles can be defined on the basis of a relation whose arguments are characterized by specific properties. For example, the role of 'being a student' can be defined as: "a student is a person enrolled in a university". In this case, 'being a student' is defined on the basis of 'being enrolled in', 'being a person', and 'being a university'. Formally, considering the previous properties as predicates, this definition can be formulated as:

$Student(x) \equiv_{df} Person(x) \wedge \exists y(enr(x, y) \wedge University(y))$

But given a specific relation r of arity n, it is possible to define n different predicates. For example, in the case of the relation $enr(x, y) \supset (Person(x) \wedge University(y))$, the predicate $EnrollingUni$ can be defined as:

$EnrollingUni(x) \equiv_{df} \exists y(enr(y, x))$

Hence the authors are aware that there is an asymmetry in the relation defining roles. $EnrollingUni$ has exactly the "same logical form" as $Student$, but this does not imply that $EnrollingUni$ is a role. Let us assume a theory containing an axiom stating that, necessarily, universities enroll at least one student, i.e., when a university loses all its students, it ceases to be a university. In this theory, ‘being an enrolling university’ is a rigid property of universities, and therefore it cannot be a role (assuming $University$ as rigid). In addition, the two predicates $EnrollingUni$ and $University$ coincide from an extensional point of view (since all universities are enrolling universities) and they cannot be distinguished by means of the theory. In this case, the predicate $EnrollingUni$ seems “redundant” with respect to the predicate $University$ because they are provably equivalent.

To extend [7]’s framework to take into account the reification of n-ary relations, [20] introduce a classification relation where a relation r is considered in the domain of quantification: $CF(x_1, \ldots, x_1, r, t)$ stands for “at the time t, the individuals $x_1, \ldots, x_n$ are classified by the relation r”. Second, they extend the primitives DF and US to the reification of predicates in general, i.e., both concepts and relations.

The fact that $Student$ and $EnrollingUni$ are concepts defined on the basis of the same relation enr is represented by the fact that $Student$, $EnrollingUni$, and enr are used in the same description d. Moreover, a link between a relation and the concepts it defines is necessary to avoid the symmetry with the other arguments of the relation. They thus introduce the predicate df, with $df(x, y)$ standing for “the (relational) concept x is defined by the relation y”. Clearly, in order to define a relational concept x, a description needs to use the relation y by which x is defined:
$(DF(x, d) \wedge df(x, y)) \supset US(y, d)$.

4 The ontology of organizations

4.1 Ontological requirements

Summarizing the discussion in Section 2, the basic properties of institutions, organizations and roles are that, first, organizations have an organizational structure in terms of sub-organizations and roles. Second, roles are defined by the organizations they belong to. The decomposition hierarchy of the organizational structure, however, is not based on the part-of relation of objects. In particular, it is transitive (a role in a department is part of the organization the department belongs to), but the parts do not exist without and before the whole. Third, there is another type of relation among the parts of an organizations, specifying which components have power on other components.

The formal framework of Masolo *et al.* [7] is the suitable starting point for defining a foundational ontology of organizations and roles. Our requirements, however, are not fully satisfied in their axiomatization.

First of all, they do not consider the structure of social entities. They do not define sub-organizations nor roles as parts of organizations. So a social entity does not have a recursive decomposition structure. Roles have been recognized as depending on some other entity which is used in their definition, but they are not defined in the entity they depend on. Moreover, we need to extend this dependence relation to specify that also sub-organizations, and not only roles, depend on the organizations.

Moreover in [7] there is no notion of power, that is the possibility that the components of an organization can affect the state of each other. However, they offer the notion of a description defining an institution, which we will use for introducing power.

The extended framework of [20] is a closer starting point for our axiomatization. The introduction of an explicit relation between an institution and a role explains the link between them. But still they do not capture the fact that a role is part of the institution and it is defined by it as we claim.

We will fulfill the above requirements in our ontology in the following way. The organizational structure of an institution is defined exploiting the fact that a social entity is defined by a description. We say that a sub-organization or a role are defined by a description which is part of the description defining the institution they belong to. This explains also why the relations associating roles to institutions are asymmetric and why roles are part of the institution and not only involved in a relation with the institution.

Concerning power, we have to model the fact that a behavior of an organization or role can access the state of another organization or role where institutional facts are represented as private behaviors or private properties. A behavior can be an action, in an agent setting, or a method, in an object oriented one which makes an institutional change; a property can be a goal, a belief, an obligation of an organization or role, *etc*. The fact that a description of an organization contains the description of a sub-organization allows the components of the organization to access each other. The idea is that all components of an organization are defined at the same time and by the same author, thus it is safe that the private methods or actions and private properties of a component can be accessed by another component's methods or actions. A behavior or a property can be accessed by a behavior not only when it is public, but also when it is private. The condition is that the accessed entity is an organization and the entity who is accessing it is a component of that organization or belongs to the same organization (e.g., when a role accesses another role). In these cases we say that the behavior is a power.

4.2 Concepts and relations

In the ontology we define the following predicates used in the definitions below:

- The predicates social concept CN and description DS are borrowed from [7]. Moreover, we need the concept of behavior *BH* and property *PROP* to model methods or actions and properties of entities, either real or social.
- The part-of relation P is extended to hold between descriptions: a description d of a concept c can use US other concepts, but it can also include the definition of another concept. We assume P is a transitive property and that a part (pre)exists independently of the whole:
  $P(a,b) \supset \exists t(PRE(a,t) \wedge \neg PRE(b,t))$
- The classification relation CF is extended as in [20] to relations; we omit the temporal index when it is not necessary.
- The relation *defined-by* relates concepts and descriptions $DF(c,d)$: the concept c ($CN(c)$) is defined by the description d ($DS(d)$). The defined-by relation is used also to define the relation *MDF* which identifies a minimal description of a concept c: a description which cannot be reduced without being unable to define the concept.
  MDF$(c,d) \equiv_{df} DF(c,d) \wedge \neg\exists d' P(d',d) \wedge DF(c,d')$

Note that to have non-minimal descriptions we have to change Axiom A8 of [7] (and thus theorem T2), so that only minimal descriptions are required to be unique: (A8') $(\mathit{MDF}(x,y) \wedge \mathit{MDF}(x,z)) \supset y = z$

- Besides describing concepts, descriptions define relations between concepts and their properties and behaviors (e.g., methods and actions). We distinguish two kinds of relations between a concept and a property or behavior: $public$ and $private$. This captures the idea usual in programming languages or in modelling languages like UML that some properties and behaviors are accessible (properties can be visible or modified, and behavior invoked) by other entities while some others are not. In Section 4.4 we show that privately accessible properties and methods play a role in the definition of powers of organizations and roles.
  Thus, in order to define accessibility we reify the two special relations *private* and *public*. $CF(c, i, \mathit{private})$ means that the concept c and the property or behavior i are classified by the *private* relation defined by description d $DF(\mathit{private}, d)$.
  $CF(c, i, \mathit{private}) \supset ED(c) \wedge (\mathit{BH}(i) \vee \mathit{PROP}(i))$
- The access relation specifies when behaviors associated to entities can access the behaviors and properties of other entities:
  $access(x, a, y, b) \supset \mathit{BH}(a) \wedge (\mathit{BH}(b) \vee \mathit{PROP}(b))$
  This access relation is expressed in terms of public properties and behaviors, but it is also defined in more complex terms when we have organizations.

4.3 The structure of organizations

The first requirement of a foundational ontology is that organizations are institutions which have a structure. We do not introduce here a primitive part-of relation between organizations and suborganizations, nor we can use P since we need different properties, like the fact that the parts do not exist without the whole. An organization c is part-of *IP* another organization c' if it is defined inside the minimal description defining the other one. Note that we need a minimal description, otherwise we could have a description d which is the union of two (minimal) descriptions d' and d'' defining two unrelated concepts. Requiring a minimal description thus means that the definition of c is essential to define c'.

$\mathit{IP}(c, c') \equiv_{\mathit{df}} \exists d, d' \; \mathit{MDF}(c, d) \wedge \mathit{MDF}(c', d') \wedge P(d, d')$

Since the P relation between descriptions is transitive, also the *IP* relation is transitive: a role which is part of a sub-organization of an organization, it is also part of the organization.

The following axiom states that if a sub-organization c is part of organization c' then the concept c' is used in the definition of c.

(B1) $\mathit{IP}(c, c') \supset \exists d \; \mathit{MDF}(c, d) \wedge US(c', d)$

We can use the *IP* predicate to define our notion of definitional foundation *DFD*. Our definition is a revised version of the founded FD predicate of [7]. It captures the idea that an instance of sub-organizations and roles is not only an instance of a concept which is part of (*IP*) another concept, but it requires the existence of an instance of such concept.

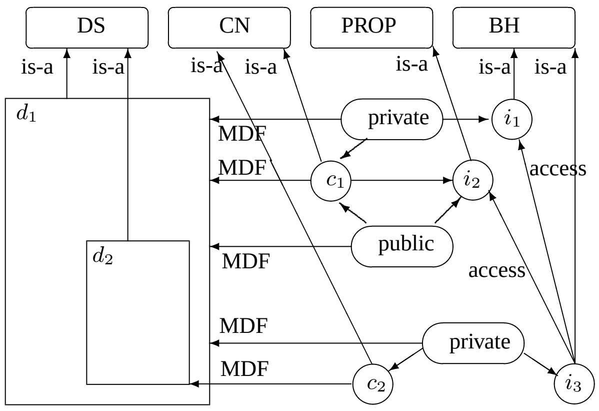


Fig. 1. An example of organization.

Definition 1 (Definitional foundation).

DFD$(x) \equiv_{df}$
$\exists y$ *IP*$(x,y) \wedge \forall z,t\,(CF(z,x,t) \supset \exists z'(CF(z',y,t) \wedge \neg P(z,z',t) \wedge \neg P(z',z,t)))$
We write also:
DFD$(x,y) \equiv_{df}$
IP$(x,y) \wedge \forall z,t\,(CF(z,x,t) \supset \exists z'(CF(z',y,t) \wedge \neg P(z,z',t) \wedge \neg P(z',z,t)))$

The difference with respect to the FD predicate of [7] is that it does not require that a concept is used in a definition of x, but that the definition is part of another concept.

Which is the relation between the two definitions? The *DFD* property is stronger than FD since we assume Axiom B1.

Theorem 1.
From Axiom B1 and from the fact that MDF$(x,d) \supset DF(x,d)$ *we have:*
DFD$(x) \supset [\exists y,d\; DF(x,d) \wedge US(y,d) \wedge$
$\forall z,t\,(CF(z,x,t) \supset \exists z'\,(CF(z',y,t) \wedge \neg P(z,z',t) \wedge \neg P(z',z,t)))] \supset FD(x)$

We can introduce now our definition of institutions, organizations and roles. Institutions are simply social concepts defined by descriptions, organizations are institutions which have sub-organizations and roles as their parts, sub-organizations are organizations which are definitionally founded on some organization and roles are anti-rigid definitionally founded concepts, and there is no institution dependent on them.

Definition 2 (Institutions, organizations and roles).

INST$(x) \equiv_{df} CN(x)$

ORG(x) $\equiv_{df}$ *INST*$(x) \wedge \exists y$ *DFD*(y,x)

S-ORG$(x) \equiv_{df}$ *ORG*$(x) \wedge$ *DFD*(x)

RL(x) $\equiv_{df} AR(x) \wedge$ *DFD*$(x) \wedge \neg\exists y$ *DFD*(y,x)

In the following example a simple organization composed by one institution with one role is illustrated:

Example 1.

c_1 is an organization which is minimally defined by description d_1 (see Figure 1). It has a private behavior i_1 and a public property i_2. Description d_1 includes also a subdescription d_2 which is the minimal description of the concept c_2, a role of c_1. c_2 has a private property i_3.

$DS(d_1)$, $DS(d_2), CN(c_1), CN(c_2), \mathit{BH}(i_1), \mathit{PROP}(i_2), \mathit{BH}(i_3) P(d_2, d_1)$
$\mathit{MDF}(c_1, d_1), \mathit{MDF}(c_2, d_2), \mathit{MDF}(d_1, \mathit{private}), \mathit{MDF}(d_1, \mathit{public}), \mathit{MDF}(d_2, \mathit{private})$[1]
$CF(c_1, i_1, \mathit{private})$, $CF(c_1, i_2, \mathit{public})$, $CF(c_2, i_3, \mathit{private})$

Thus, c_2 is a part of c_1: $\mathit{IP}(c_2, c_1)$

In [6] we consider another definitional property of roles: the fact that a role can play a role. This property is implicit in the fact that nothing prevents that a role, as a social concept, can be classified by another role. Note that this is in contrast with the position about role playing roles stated in [7], even if their model allows also the alternative we choose.

4.4 Powers

Properties and behaviors associated with organizations cannot be all freely accessed by any agent. Some of them, e.g., the building where an organization is officially located, are physical properties which every agent can manipulate. In contrast, other properties have only a social character, and thus are immaterial: the id number of the employees, the action of firing an employee, making the organization buy some goods, obliging an employee to do something, changing the structure of the organization. Since these institutional properties and behaviors are immaterial, how can they be manipulated? As discussed in Section 2 institutional properties are controlled by counts as rules. In our ontology we model counts as rules defining powers as behaviors of social entities (organizations and roles) which access properties. To represent the fact that institutional properties can be manipulated only from inside an institution we model them as private properties of institutions.

The problem to be solved is the behaviors of which entities can access a private property of an institution, since the visibility rules in organizations are different than in objects. The notion of power is thus based on the definition of an $access$ relation defining a sort of scope for behaviors and properties.

We do not describe here behaviors. For example, actions could be described by plan operators and methods by programs. We represent, however, that a behavior accesses other behaviors (since they are actions in a plan or invoked by a program) or some properties (the value of the property is needed for executing the behavior or it is changed by the behavior).

[1] This does not mean that *public* is defined twice, but that its extension (i.e., the tuples of entities classified CF by it) is determined by both descriptions.

An entity classified by a concept c can access a property p of another entity, if it is a public property, or it is a private property of a concept c' whose definition defines also the concept c (i.e., c is part of *IP* c') or of another concept which depends on c' too.

Definition 3 (Powers).

The access relation is defined as:

$access(x, a, y, b) \equiv_{df} public(y, b) \vee superaccess(x, y, b) \vee peeraccess(x, y, p)$

A method or property b of individual y is public if it is a public behavior or property of a concept subsuming y:

$public(y, b) \equiv_{df} \exists c\, CN(c) \wedge CF(y, c) \wedge CF(c, b, \textit{public})$

A method or property b of individual y can be accessed from individual x if the concept subsuming x is part of another concept subsuming y (or vice-versa) and b is a private behavior or property of the latter.

$superaccess(x, a, y, b) \equiv_{df} \exists c, c'\, CF(x, c) \wedge CF(y, c') \wedge \textit{IP}(c, c') \wedge CF(c', b, \textit{private})$

A method or property b of individual y can be accessed from individual x if the concept subsuming x is part of another concept which has as its part a concept subsuming y and b is a private behavior or property of this concept.

$peeraccess(x, a, y, b) \equiv_{df} \exists c, c', c''\, CF(x, c) \wedge CF(y, c') \wedge \textit{IP}(c, c'') \wedge \textit{IP}(c', c'') \wedge CF(c', b, \textit{private})$

A behavior of an entity is a power if it can $superaccess$ or $peeraccess$ another behavior.

$\textit{POW}(x, a) \equiv_{df} \exists y\, superaccess(x, a, y, b) \vee peeraccess(x, a, y, b)$

If we impose that *IP* is a reflexive relation, then we have that a behavior of an entity can access the private behaviors and properties of itself and that the behaviors of an organization can access the private state of its components.

Note that the $access$ relation specifies which behaviors can access other behaviors and properties. This does not mean that in an actual organization every behavior accesses every other behaviors or properties. The fact that a behavior accesses some other behavior or property depends on how this behavior is defined in the description by means of plans or programs. As we said, the author of the definition of the organization is the author of the definitions of its components, so the access definition is safe. Nothing prevents, however, that a more restrictive definition of access is given to respect the organizational structure. For example, it can be defined on a non-transitive part-of relation, so that each component can have powers only on its direct super or sub components or on its siblings.

Example 2. In Figure 1 behavior i_3 can access both behavior i_1 and property i_2 even if the former is private, since *IP*(c_2, c_1).

Note that dealing with visibility rules in a programming languages is a complex issue. In this model we do not want to propose to define a general notion of accessibility, but to study the peculiarities of accessibility in organizations.

5 Applications

In this section we explain how the foundational ontology presented here matches our previous work, and, in particular, how it can be used to introduce organizations and roles not only in multiagent systems but also in the object oriented paradigm.

We study normative systems [15, 22] and organizations composed of sub-organizations and roles [5, 6] using the so called agent metaphor. The agent metaphor allows to describe social entities, like normative systems, as they were agents, and thus attributing them mental attitudes like beliefs and goals. What corresponds in the agent metaphor to the basic primitives of our foundational ontology? First of all, we have to explain the structure of an organization. An agent does not have parts which are agents themselves, so an organization-as-an-agent cannot have other organizations as its parts. Rather, to structure organizations we exploit the idea that an agent can attribute mental attitudes to other entities via the agent metaphor, also to entities which are not agents. Since an organization is described as an agent, then it can attribute mental attitudes to other entities. In this way, it can define sub-organizations and roles by describing them as agents, in a recursive way.

As Searle claims, social entities are defined by means of constitutive (and regulative) rules. In [15]'s model beliefs attributed to a social entities correspond to the constitutive rules and goals the regulative rules. Thus, describing a social entity as an agent amounts to defining it. A definition of a sub-organization is included in the definition of the organization it belongs to since, in the definition of the latter are present not only the beliefs and goals attributed to them, but also the beliefs and goals which it attributes to sub-organizations and roles.

Like in our foundational ontology, powers arise from the fact that all the structure of the organization is defined in the same definition: so that the constitutive rules of a sub-organization can refer to other sub-organizations as well.

Even if at first sight can be surprising, our foundational ontology of organizations can be used to model organizations by means of standard object oriented representation languages, like UML. Rather than adding primitives to UML, we use a pattern. This does not mean that it is not useful to introduce some primitives which are based on this pattern. As a consequence, institutions can be introduced also in object oriented programming languages like Java. Thus, in [23, 24] we present an extension of Java, called powerJava, where suitable constructs are introduced to represent roles.

The basic idea is that the description of a concept in object orientation corresponds to a class and in UML and some programming languages a class can contain other classes, called inner classes. Outer classes correspond to descriptions having other descriptions as parts. An inner class can contain further inner classes as well, thus allowing a recursive decomposition structure. Moreover, inner classes have the features we need for modelling institutions: dependence and powers. First, an instance of an inner class does not exist without an instance of the outer class, since it has a reference to an instance of the outer class. Second, the methods of an inner class can access the state of the outer class and of other sibling inner classes. Powers thus can be modelled just as the methods of an inner class.

The difference between sub-organizations and roles is that roles do not have further inner classes inside them and that they are associated to a player via a reference. Roles are anti-rigid because they are associated to their players by a reference, rather than being modelled as sub or super classes (like other proposals for representing roles, e.g., [25], do instead). Thus an inner class representing a role has always two references to two objects: the institution that defines it and the player that plays it.

6 Conclusions

In knowledge representation, and more specifically in the field of description logics, the term 'role' is nowadays synonymous of an arbitrary binary relation (often a function) used to characterize the structure of a concept. The concept 'person', for instance, may have the role 'likes', which represents the relationship between a person and what she likes best. But this is not what is meant by social roles.

In multi-agent systems (MAS) roles are generally viewed as descriptions of agent's acting and interacting, where agents include also societies or organizations of agents. The characterization of this kind of social roles (in the restricted sense) is founded on theories of action and behavior (involving tasks, goals, plans, *etc.*) and deontic notions. In [2] a role is viewed as an "abstract description of an entity's expected function" which is defined by four attributes: responsibilities (that determine the functionality of the role), permissions, activities, and protocols. Pacheco and Carmo [26] clearly distinguish roles from agents (agents can act, and roles cannot). But these descriptions do not tell much about what distinguish roles from objects or agents.

In object-oriented programming languages the focus has been on technical issues (multiple and dynamic classification, multiple inheritance, objects changing their attributes and behaviors, *etc.*), rather than what are the roles' distinguishing properties.

In this paper we propose a foundational ontology of organizations and roles which extend Masolo *et al.* [7]'s proposal. Institutions are social concepts which exist because of descriptions defining them, which are collectively accepted. Organizations are institutions which have a structure in terms of sub-institutions. Sub-organizations are organizations which are parts of other organizations. Finally, roles are components of organizations which do not have further organizational structure and which can be played by agents.

This work builds on our previous work on normative multiagent systems and organizations based on the agent metaphor. In [4] we present the agent metaphor to build a cognitive ontology. Here, instead we present an axiomatic ontology built in an analytical style. This work aims at isolating the essential properties which distinguish social concepts from other kind of entities and to justify the choices made in previous works. Moreover, we show that this ontology can be used to extend current representation languages like UML and object oriented programming languages.

References

1. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: an organizational view of multiagent systems. In: LNCS n. 2935: Procs. of AOSE'03, Springer Verlag (2003) 214–230
2. Zambonelli, F., Jennings, N., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. IEEE Transactions of Software Engineering and Methodology **12(3)** (2003) 317–370

3. Searle, J.: The Construction of Social Reality. The Free Press, New York (1995)
4. Boella, G., van der Torre, L.: An agent oriented ontology of social reality. In: Procs. of FOIS'04, Amsterdam, IOS Press (2004) 199–209
5. Boella, G., van der Torre, L.: Organizations as socially constructed agents in the agent oriented paradigm. In: LNAI n. 3451: Procs. of ESAW'04, Berlin, Springer Verlag (2004) 1–13
6. Boella, G., van der Torre, L.: The ontological properties of social roles: Definitional dependence, powers and roles playing roles. In: Procs. of LOAIT workshop at ICAIL'05. (2005)
7. Masolo, C., Vieu, L., Bottazzi, E., Catenacci, C., Ferrario, R., Gangemi, A., Guarino, N.: Social roles and their descriptions. In: Procs. of KR'04, AAAI Press (2004) 267–277
8. Jennings, N.R.: On agent-based software engineering. Artificial Intelligence **117(2)** (2000) 277–296
9. Yu, E.: Agent orientation as a modelling paradigm. Wirtschaftsinformatik **43(2)** (2001) 123–132
10. Dastani, M., van Riemsdijk, B., Hulstijn, J., Dignum, F., Meyer, J.J.: Enacting and deacting roles in agent programming. In: Procs. of AOSE'04, New York (2004)
11. Grossi, D., Dignum, F., Dastani, M., Royakkers, L.: Foundations of organizational structures in multiagent systems. In: Procs. of AAMAS'05. (2005)
12. McCallum, M., Norman, T., Vasconcelos, W.: A formal model of organisations for engineering multi-agent systems. In: Procs. of CEAS Workshop at ECAI'04. (2004)
13. Booch, G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, Reading (MA) (1988)
14. Searle, J.: Speech Acts: an Essay in the Philosophy of Language. Cambridge University Press, Cambridge (UK) (1969)
15. Boella, G., van der Torre, L.: A game theoretic approach to contracts in multiagent systems. IEEE Transactions on Systems, Man and Cybernetics - Part C (2006)
16. Sowa, J.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks/Cole, Pacific Growe (CA) (2000)
17. Guarino, N., Welty, C.: Evaluating ontological decisions with ontoclean. Communications of ACM **45(2)** (2002) 61–65
18. Fine, K.: Ontological dependence. Proceedings of the Aristotelian Society **95** (1995) 269–290
19. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Schneider, L.: Sweetening ontologies with dolce. In: Proc. EKAW 2002, Siguenza (SP) (2002)
20. Masolo, C., Guizzardi, G., Vieu, L., Bottazzi, E., Ferrario, R.: Relational roles and qua-individuals. In: Procs. of AAAI Fall Symposium Roles'04, AAAI Press (2005)
21. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering **35** (2000) 83–848
22. Boella, G., van der Torre, L.: Security policies for sharing knowledge in virtual communities. IEEE Transactions on Systems, Man and Cybernetics - Part A (2006)
23. Baldoni, M., Boella, G., van der Torre, L.: Bridging agent theory and object orientation: Importing social roles in object oriented languages. In: Procs. of PROMAS'05 workshop at AAMAS'05. (2005)
24. Baldoni, M., Boella, G., van der Torre, L.: Roles as a coordination construct: Introducing powerJava. In: Procs. of MTCoord'05 workshop at COORDINATION'05. (2005)
25. Albano, A., Bergamini, R., Ghelli, G., Orsini, R.: An object data model with roles. In: Procs. of VLDB'93. (1993) 39–51
26. Pacheco, O., Carmo, J.: A role based model of normative specification of organized collective agency and agents interaction. Autonomous Agents and Multiagent Systems **6** (2003) 145–184

When Agents Communicate Hypotheses in Critical Situations

Gauvain Bourgne[1], Nicolas Maudet[1], and Suzanne Pinson[1]

LAMSADE, Université Paris-Dauphine
Paris 75775 Cedex 16 (France)
Email: {bourgne,maudet,pinson}@lamsade.dauphine.fr

Abstract. This paper discusses the problem of *efficient propagation of uncertain information* in dynamic environments and critical situations. When a number of (distributed) agents have only partial access to information, the explanation(s) and conclusion(s) they can draw from their observations are inevitably uncertain. In this context, the efficient propagation of information is concerned with two interrelated aspects: spreading the information as quickly as possible, and refining the hypotheses at the same time. We describe a formal framework designed to investigate this class of problem, and we report on preliminary results and experiments using the described theory.

1 Introduction

Consider the following situation: witness of a threathening and unexpected event, say a fire in a building, Jeanne has to act promptly to both escape the danger and warn other people who might get caught in the same situation. However, there are no official signs or alarms indicating where the fire actually started: Given her partial knowledge of the situation, Jeanne may build some hypotheses explaining her observations (where the fire did start in the first place, maybe why), but the conclusions she may reach would remain *uncertain*. (That is, uncertainty here lies on the fact that she has incomplete knowledge of the world, rather than untrusted perceptions of this world). In addition, there is no way for Jeanne to trigger an alarm. In other words, Jeanne will try to both circulate the information in order to spread the information to colleagues, and refine the hypothesis at the same time. Typically, Jeanne faces two questions:

- What information should I transmit?
- To whom should I transmit this information?

Clearly, these two questions are interelated. Depending on the person Jeanne selected to communicate with, she may decide to transmit different messages: the objectives being to ensure that the transmitted information can be used efficiently in the next transmission, and so on. This defines, we believe, a problem of efficient propagation of uncertain information. The purpose of this paper is to put forward a formal framework expliciting both the reasoning and communicational aspects involved in these situations. We explore some preliminary properties of the proposed framework and interaction protocol, and illustrate our approach with a case study experimented using the described theory.

The remainder of this paper is as follows. Section 2 presents the formal reasoning machinery that we shall use in the framework: it heavily builds upon Poole's Theorist system [14]. Section 3 details the communication module, and explores specifically some properties of a protocol designed to exchange hypothesis. Section 4 describes our case study example, instantiating the proposed framework. The situation involves a number of agents trying to escape from a burning building. We give the detail of a simple example, showing how critical, in this crisis context, can be the decisions taken by agents as to whether/what communicate. Section 5 draws connections to related works, and Section 6 concludes.

2 Agents Reasoning

This section introduces the formal machinery involved in the agents reasoning process. The described situation suggests agents able to deal with partial perception of the world, to build hypotheses from observations they make, to draw conclusions from a set of explanations, and to communicate with each other in order to exchange pieces of information. Agents reasoning process builds on Poole's framework [14], which allows to elegantly combine both the explanation and the prediction processes, using a single axiomatization. By formulae we mean well-formed formulae in a standard first order language. Each agent is (slightly modified version) of an instance of a Theorist system [14]:

$$\langle \mathcal{F}, \mathcal{H}, C, O, E, \leq \rangle$$

where

- $\mathcal{F}$ a set of *facts*, closed formulae taken as being true in the domain
- $\mathcal{H}$ a set of formulae which act as *possible hypotheses*, common to all agents
- C a set of closed formulae taken as *constraints*, common to all agents
- O is a set of grounded formulae representing the *observations* made so far by the agent. Each agent believes every observation in this set to be true.
- E is the set of *preferred explanations*, it is the set of all justifiable explanations of the observation set O
- $\leq$ is the *preferrence relation*, a pre-order on the explanations common to all agents

We first recall a number of basic definitions.

Definition 1 (Scenario [14]). *A scenario of $(\mathcal{F}, \mathcal{H})$ is a set $\theta \cup \mathcal{F}$ where θ is a set of ground instances of elements of $\mathcal{H}$ such that $\theta \cup \mathcal{F} \cup C$ is consistent.*

In the following, we shall also refer to the conjunction h of the elements of θ as the *hypothesis* associated to this scenario.

Definition 2 (Explanation of a closed formulae [14]). *If g is a closed formula, then an explanation of g from $(\mathcal{F}, \mathcal{H})$ is a scenario of $(\mathcal{F}, \mathcal{H})$ that implies g.*

We now introduce a couple of further notions that proved to be appropriate in our context. Events occuring in the world and observed by the agents may or may not be explained, or contradicted, by the agent model.

Definition 3 (Positive observation). *A positive observation of $(\mathcal{F}, \mathcal{H})$ is an observation $o \in O$ such that there exists an explanation of o from $(\mathcal{F}, \mathcal{H})$*

Definition 4 (Negative observation). *A negative observation of $(\mathcal{F}, \mathcal{H})$ is an observation $o \in O$ such that there exists an explanation of $\neg o$ from$(\mathcal{F}, \mathcal{H})$*

In the following, we shall note $P(O)$ to refer to the set of all positive observations of $(\mathcal{F}, \mathcal{H})$, and $N(O)$ to refer to the set of all negative observations of $(\mathcal{F}, \mathcal{H})$. Note that this is not necessarily a partition: some observations may have no explanation, while some others may have both positive and negative explanations.

Definition 5 (Explanation of an observation set). *If O is a set of observations, an explanation of O from $(\mathcal{F}, \mathcal{H})$ is an explanation ξ of $P(O)$ such that $\xi \cup C \cup N(O)$ is consistent (which implies the consistency of $\xi \cup C \cup O$).*

Definition 6 (Justifiable explanation). *A justifiable explanation of O from $(\mathcal{F}, \mathcal{H})$ is an explanation such that if any element of its associated hypothesis set θ is removed from it, it is no longer an explanation of O.*

Based on this system, we also define, for each agent a_i:

1. H_i, the set of *preferred hypotheses* associated with E_i, the set of justifiable explanations. For a given set of observation O_i, $\mathcal{E}_{exp}$, the *explanation function* returns the set of all justifiable explanations of O_i from $(\mathcal{F}, \mathcal{H})$. $\mathcal{E}_{hyp}(O_i)$ gives the set of hypotheses associated with $\mathcal{E}_{exp}(O_i)$. We assume $\mathcal{E}_{exp}$ and $\mathcal{E}_{hyp}$ to be deterministic, and common to all agents.
2. h is the *favoured hypothesis* from E. The agent choses one favoured hypothesis among its own minimal hypothesis according to the preferrence relation.

In summary, for each agent we have:

- $E_i = \mathcal{E}_{exp}(O_i)$
- $H_i = \mathcal{E}_{hyp}(O_i)$
- $h_i \in min(H_i)$

This ensures that h_i is associated with a minimal justifiable explanation for O_i, that is :

- h_i is consistent with O_i, that is $\nexists o_i \in O_i$ s.t. $h_i \models \neg o_i$
- h_i explains all elements of $P(O_i)$

– h_i is justifiable from O_i, that is for each clause c_k of the conjunction h_i ($h_i = h'_i \wedge c_k$), there is an element o of $P(O_i)$ such that $h_i \models o$ but $h'_i \not\models o$.
– h_i is minimal according to the preorder $\leq$

Typically, as suggested by the aforementioned model, different explanations will exist for a given formula. What should be the preference relation between explanations? Clearly there can be many different ways to classify prefered explanations. In [14], different comparators are introduced. In our framework, we shall use variants of two of them:

1. *minimal explanation*— prefer the explanations that make the fewest (in terms of set inclusion) assumptions. In other words, no strict subset of a minimal explanation should also be an explanation.
2. *least presumptive explanation*— an explanation is less presumptive than another explanation if it makes fewer assumptions (in terms of what can be implied from this explanation together with the facts)

Now we need to see how these agents will evolve and interact in their environment. In our context, agents evolve in a dynamic environment, and we classicaly assume the following *system cycle*:

1. *Environment dynamics*: the environment evolves according to the defined rules of the system dynamics
2. *Perception step* : agents get perceptions from the environment. These perceptions are typically partial (*e.g.* the agent can only see a portion of the map), but we assume that they are *certain*, in the sense that the sensors are assumed perfect.
3. *Reasoning step*: agents compare perception with predictions, seek explanations for (potential) difference(s), refine their hypothesis, draw new conclusions. More precisely, during this step, if the agent perception prove its hypothesis false, the agent computes the possible explanations for these new perception, given its previous perception. It makes use of Theorist for this task. It must then select the action to be executed in the next phase.
4. *Action step*: agents modify the environment by executing the action selected by the previous deliberation steps.

What remains to be described, of course, is the interaction module and the way agents will exchange hypotheses and observations.

3 Agent Communication

In our system, observations are not only made directly by agents (by perceiving the environment): they can also result from communication between agents. The cycle is then augmented with an explicit *communication step*, which directly follows the *reasoning step*. During the *Communication step*, agents engage communication with other agents to warn of their observation and tune up their hypothesis. In a given round, a given agent can only communicate with *one* agent. If that agent is occupied talking to another agent, it must wait or choose a different agent to communicate with. We now describe the interaction protocol pictured in Fig. 1, together with agents' behaviour.

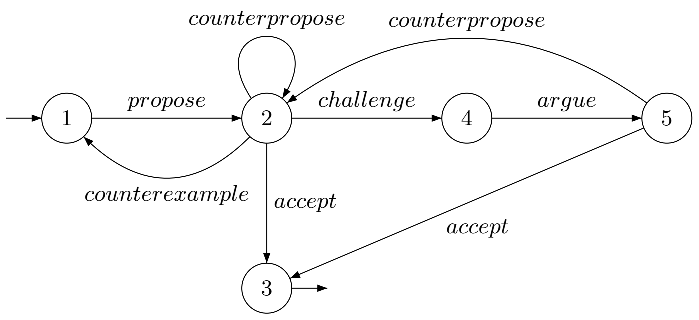


Fig. 1. Hypotheses Exchange Protocol.

3.1 Description of the Interaction Protocol and Strategies

Upon receiving a hypothesis h_1 (*propose*($h1$) or *counterpropose*($h1$)) from a_1, agent a_2 is in state 2 and has the following possible replies:

- if $\exists o_2 \in N(O2)$ s.t. $h_1 \models \neg o_2$, then the agent knows a counter-example that contradicts this hypothesis: he will communicate this counter-example and utter *counterexample*(o_2). We are back in state 1 of the protocol. Agent will then recompute his hypothesis with this new fact, and will propose h_1'.
- if $\exists o_2 \in P(O2)$ s.t. $h_1 \not\models o_2$, then the agent knows an example of positive observation that is *not* explained by this hypothesis: he will communicate this uncovered example and utter *counterexample*($o2$), as in the previous case.
- otherwise, no observation made by a_2 contradicts h_1 and h_1 implies $P(O_2)$, that h_1 is the hypothesis associated with an explanation of $O2$. We have then the following cases:
  - if the agent has no argument in favour of the hypothesis ($h1 \notin H_2$ where H_i is the set of the hypothesis associated to agent a_i's preferred explanations), he will *challenge* a_1 in order to obtain some arguments supporting this hypothesis. Agent a_1 is then bound to communicate an argument (*argue*(arg))[1], leading to state 5. Upon receiving this argument, a_2 recomputes his hypothesis by using this argument. If h_1 is obtained, he will *accept*, leading to the final state 3. Otherwise, a different hypothesis h_2' is obtained and proposed, leading back to state 2.

[1] Note that the agent keeps track of the communicated arguments, which allows him not to send twice the same argument to this agent during a communication step.

- otherwise $h_1 \in H_2 : h_1$ is a hypothesis associated to a justifiable explanation of O. We have then two possibilities:
  - $*$ if h_1 is not preferred to h_2 in the sense of the defined preference relation, then agent a_2 would *counterpropose*(h_2), leading to state 2 with inverted roles.
  - $*$ otherwise, h_1 is necessarily prefered to h_2: a_2 will then respond *accept*, concluding the conversation (state 3).

3.2 Local Properties of the Interaction Protocol

We first investigate locally the properties of the proposed protocol, that is, the outcome of a single dialogue governed by the rules and decision process described in the previous subsection, and involving only two agents.

Lemma 1. *Let $c = |O_1 \cup O_2| - |O_1 \cap O_2|$. If $c = 0$ then $O_1 = O_2$ and $H_1 = H_2$.*

Proof. Clearly, $O_1 \cap O_2 \subseteq O_1 \cup O_2$. If $c = 0$, $|O_1 \cup O_2| = |O_1 \cap O_2|$, hence $O_1 \cup O_2 = O_1 \cap O_2$. Now because $O_1 \cap O_2 \subseteq O_1 \subseteq O_1 \cup O_2$, (and symetrically for O_2), we have $O_1 = O_2$. By virtue of the determinism of the explanation function, we conclude that $H1 = \mathcal{E}_{hyp}(O_1) = \mathcal{E}_{hyp}(O_2) = H_2$. $\square$

The first property that needs to be verified is the *termination*. We show that this algorithm enjoys this property.

Property 1 (Termination). Termination is guaranteed, and the length of the interaction process (in terms of the number of exchanged messages) is bounded by $4 \times c + |O_1 \cap O_2|$.

Proof. Let $c = |O_1 \cup O_2| - |O_1 \cap O_2|$. By Lemma 1, we know that in case $c = 0$, it follows that $O_1 = O_2$ and $H_1 = H_2$ (in which case we note $O = O_1 = O_2$ and $H = H_1 = H_2$). Then observe that, $H = \mathcal{E}_{hyp}(O)$, together with the fact that $h_1, h_2 \in H$, guarantees that h_1 and h_2 are the favored hypotheses of the justifiable explanations of O. The following points then follow (i) $\nexists o \in O$ s.t. $h_1 \models \neg o$ or $h_2 \models \neg o$, (ii) $\nexists o \in P(O)$ s.t. $h_1 \not\models o$ or $h_2 \not\models o$, (iii) $h_1 \in H_2$ and $h_2 \in H_1$, and (iv) both $h_1, h_2 \in min(H)$, no hypothesis is then strictly prefered to the other one.

Given this, as soon as the system is in state 2, all termination conditions are met. But we also know that the message exchange between agents leads to state 2 every 3 messages at most. Termination is then guaranteed when $c = 0$.

We now need to prove that c will eventually reach the value 0. To do that, we will show that every 4 messages at most, it decreases of 1.

The first message leads to state 2. Without loss of generality, we assume that the last message is, say, from agent a_j to agent a_i (hypothesis h_j is then proposed to a_i). Following the agent's decision algorithm previously described, there are now four possibilities:

(i) $\exists o_i \in O_i$, s.t. $h_j \models \neg o_i$ or $\exists o_i \in P(O_i)$, s.t. $h_j \not\models o_i$, then a_i sends a counterexample o_i to a_j. In this case, $O'_j = O_j \cup \{o_i\}$ with $o_i \in O_i$ and $o_i \notin O_j$, which means that $|O'_j \cap O_i| = |O_j \cap O_i| + 1$, and $|O_j \cup O_i|$ remains unchanged. It follows that c is decreased by 1.

(ii) $\not\exists o_i \in O_i$ s.t. $h_j \models \neg o_i$ and $\forall o_i \in P(O_i)$, $h_j \models o_i$ and $h_j \notin H_i$, then a_i requires an argument and a_j provides o_j. In this case, $O'_i = O_i \cup \{o_j\}$. If $o_j \in O_i$, then a_i repeats its challenge until he gets an observation o_j he didn't know before. Since a_j keeps track of its messages, at most $|O_1 \cap O_2|$ such messages can be exchanged. We eventually reach o'_j such that $O'_i = O_i \cup \{oj'\}$ where $o'_j \in O_j$ and $o'_j \notin O_i$.

(iii) $\not\exists o_i \in O_i$ s.t. $o_i \models \neg h_j$ and $\forall o_i \in P(O_i)$, $h_j \models o_i$ and $h_j \in H_i$ but $h_j \notin min(H_i)$, then a_i respond with *counterpropose*(h_i). We are back in state 2, but now we are sure that $h_i \notin H_j$ (because $h_i \leq h_j$ and $h_j \in min(H_j)$, by definition), which means that we would be in case (i) or (ii).

(iv) $\not\exists o_i \in O_i$ s.t. $o_i \models \neg h_j$, and $\forall o_i \in P(O_i)$, $h_j \models o_i$, and $h_j \in min(H_i)$, but then a_i accepts and the protocol terminates.

□

Corollary 1. *After termination, the following properties are guaranteed:*

- *a_1 and a_2 are consistent*
- *a_1 and a_2 have a hypothesis that explains both $P(O_1)$ and $P(O_2)$*
- *a_1 and a_2 have a hypothesis that is justifiable from O_1 and O_2*
- *a_1 and a_2 have a hypothesis that is minimal for O_1 and O_2 (that is $h_1 \in min(\mathcal{E}_{hyp}(O_2))$ and $h_2 \in min(\mathcal{E}_{hyp}(O_1))$)*

3.3 Global Properties of the Communication Protocol

The properties previously described hold locally, when only two agents interacts over one communication step. The next question is then to ask whether these properties can be guaranteed at a more global level. Clearly, many properties will not hold any longer when considered globally. One simple such property is the consistance, which cannot be transitive when only based on the bilateral hypothesis exchange protocol described. This can be observed by constructing an example where an agent a would first communicate a hypothesis to agent b, not revealing the full arguments supporting its position though. Now if b communicates in turn with a third agent, say c, it is clear that he may not be in a position to effectively defend this hypothesis, and may accepting c's hypothesis. a and c would then not be consistent. This is formally stated as follows.

Property 2. The consistance property guaranteed by the communication protocol is not transitive.

Proof. We construct the following counterexample : agent a_1 can communicate with agent a_2 and a_3, but agents a_2 and a_3 cannot communicate with each other. We assume that they share the following facts $\{p(X) \rightarrow r(X), q(X) \rightarrow$

$r(X), p(X) \to s(X)\}$, where $p(X)$ and $q(X)$ are hypothesis. We start with the following sets of observations $P_1 = \{r(a), \neg p(X)\}$, $P_2 = \{\,\}$, and $P_3 = \{s(a)\}$. Agent a_1 communicates $q(A)$, which is challenged by a_2. a_1 then provides an explanation ($r(a)$). Now a_2 communicates with a_3 and proposes $q(a)$, but a_3 has an additional observation, namely $s(a)$. Upon receiving this hypothesis, a_3 challenges a_2 and a_2 provides the only argument he has in possession: $r(a)$. But a_3 knows the further observation that $s(a)$ which makes the hypothesis $p(a)$ prefered. a_3 makes this counterproposal, a_2 challenges and a_3 gives his argument ($s(a)$). Now a_2 will accept. At this point of the interaction though, a_1 holds $q(a)$ as favoured hypothesis, while a_3 prefers $p(a)$, which is not consistent with $\neg p(X) \in P_1$. □

What this suggests is that we will need much more elaborated synchronization techniques to guarantee that these desirable properties still hold at the global level. However, in our context where time is a critical factor, and where communication can be highly restricted, it will be interesting to investigate in which situations simple protocols, like the one described here, can still give promising result and ensure an average good efficiency of the information propagation. As a first step towards this objective, we give in the next section an instance of the proposed framework and show a critical situation where communication and hypothesis exchange proves to be efficient.

4 A Case Study: Crisis Management

This section presents an instance of the general framework introduced earlier. We first describe the different parameters used to instantiate the framework. A complete example is then detailed.

4.1 Description of the situation

This experiment involves agents trying to escape from a burning building. The environment is described as a spatial grid with a set of walls and (thankfully) some exits. Time and space are considered discrete. Time is divided in rounds.

Agents are localised by their position on the spatial grid. These agents can move and communicate with other agents. In a round, an agent can move of one cell in any of the four cardinal directions, provided it is not blocked by a wall. In this application, agents communicate with any other agent (but, recall, a single one) given that this agent is in view, and that they have not yet exchanged their current favored hypothesis. Note that this spatial constraint on agents' communication could be relaxed in other contexts (which would require, in turn, to apply a more elaborated recipient choice algorithm).

At time t_0, a fire erupts in theses premises. From this moment, the fire propagates. Each round, for each cases where there is fire, the fire propagates in the four directions. However, the fire cannot propagate through a wall. If the fire propagates in a case where an agent is positionned, that agents burns and is considered dead. It can of course no longer move nor communicate. If an agent gets to an exit, it is considered saved, and can no longer be burned. It still can communicate, but need not move.

Agents know the environment and the rules governing the dynamics of this environment, that is, they know the map as well as the rules of fire propagation previously described. They also locally perceive this environment, but cannot see further than 3 cases away, in any direction. Walls also block the line of view, preventing agents from seeing behind them. Within their sight, they can see other agents and whether or not the cases they see are on fire. All these perceptions are memorised.

In order to deliberate, agents maintain a list of their possible explanations E (and a list of associated hypotheses H) explaining their observations about fire, and a prediction of fire propagation based on their favoured hypothesis h. The preference relation ($\leq$) is the following:

- the agent prefers the minimal explanation, taking into account only fire origins. In other words, an agent will prefer an explanation using an unique fire origin propagating over one using several sources.
- the agent prefer the least presumptive explanation, taking into account propagation and origins. In effect it means that the agent will favor an explanation considering the fire origin as closer to the observed manifestation.

Based on the reasoning described above, agents also maintain a list of possible escape route, sorted by simply favouring the shortest paths to exits.

4.2 Sample of Agents Theories

We now give a snapshot of the declarative representation of agents' knowledge, illustrating the different kind of rules involved in this example.

- Facts ($\mathcal{F}$) allow to represent the static elements of the environment, as well as the rules governing the dynamic of the environment. For instance, the following three rule state that there is indeed a vertical wall at location (0,1), that the fire can always be assumed to have started at the location it is observed, and eventually that the fire should propagate in four possible directions. This last one is an example of a rule justified in normal circumstances, but which may suffer exceptions: it is then represented as a default rule.

```
fact vwall(at(0,1)).
fact fire(T,at(X,Y)) <- origin(T,at(X,Y)).
default rule_propagates_L(T2,from(X2,Y)): fire(T,at(X,Y)) <-
        previous(X,X2), previous(T,T2), fire(T2,at(X2,Y)).
```

- The possible hypotheses set ($\mathcal{H}$), in this example application, is the set of all conjunctions of possible fire origin(s).
- Constraints (C) prevent default rules from applying. For example, the landscape includes walls and doors which prevent the fire from propagating.

```
constraint not rule_propagates_L(T,from(X,Y)) <- vwall(at(X,Y))
```

– Observations (O) can either be of the form `fire(T,at(X,Y))`, or of the form `nofire(T,at(X,Y))`

4.3 Example

We are now in a position to describe the steps of our illustrative example.

[Round t=0] A fire erupts at (6,6), but nobody can initially see it. It will propagate until t=3 before beeing seen.

[Round t=3]

Perception step. Agent a_1 sees fire at (3,6) (not expected), and agent a_3. Agent a_2 sees fire at (6,3) and (5,4) (not expected). Agent a_3 sees a_1.

Explanation step (a_1). Having computed an explanation for `fire(t=3, at (3,6))`, a_1 gets 12 possible explanations, each one exhibiting a single origin. One such explanation, as provided by the Theorist system, states that the fire may have started at location (4,5), before propagating to the north (i.e. from south) and to the west.

```
Answer is fire(t3, at(3, 6))
Theory is
[rule_propagates_R(t2, from(4, 6)),
rule_propagates_D(t1, from(4, 5)),
origin(t1, at(4, 5))]
```

To classify these hypothesis, he first selects the minimal hypothesis considering only the origin. In this case, all the hypothesis suppose only one origin for the observed fire. Among those, he then selects the less presumptive hypothesis. In this case, the selected hypothesis is:

```
[origin(t3, at(3, 6))]
```

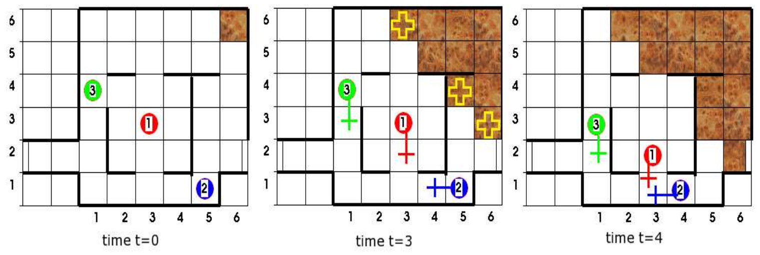
Explanation step (a_2)**.** Searching explanations for fire at (6,3) and (5,4), a_2 gets 6*6 possible explanations, such as :

```
Answer is fire(t3, at(6, 3)) and fire(t3, at(5, 4))
Theory is
[rule_propagates_R(t2, from(6, 4)),
origin(t2, at(6, 4)),
origin(t3, at(6, 3))]
```

Among those theories, only four of the explanations propose a common origin, and as such are minimal according to the origin criteria. Among those four, the less pre-emptive one is eventually:

```
[rule_propagates_R(t2, from(6, 4)),
rule_propagates_D(t2, from(6, 4)),
origin(t2, at(6, 4))].
```

Communication step. Agents a_1 and a_3 are the only agent seeing each other. Agent a_3 has no reason to initiate a communication, but a_1 has one: it has just changed its hypothesis and will try propagating and validating it. a_3 asks for arguments and a_1 sends it `fire(t=3,at(3,6))`. With this facts, Agent 3 recomputes its hypothesis and get the same favoured hypothesis. The hypothesis is confirmed and the communication stopped.



[Round t=4]

Action step. a_3 moves towards the west exit, which is the closest exit. a_1 moves towards the east exit, for the same reason. Although it is closer to the east exit, a_2 moves towards the west exit because it predicts that fire will arrive at the east exit before it can go out this way.

Perception step. Agent a_1 sees a_2 and conversely. All the fire seen by agents were predicted during this step.

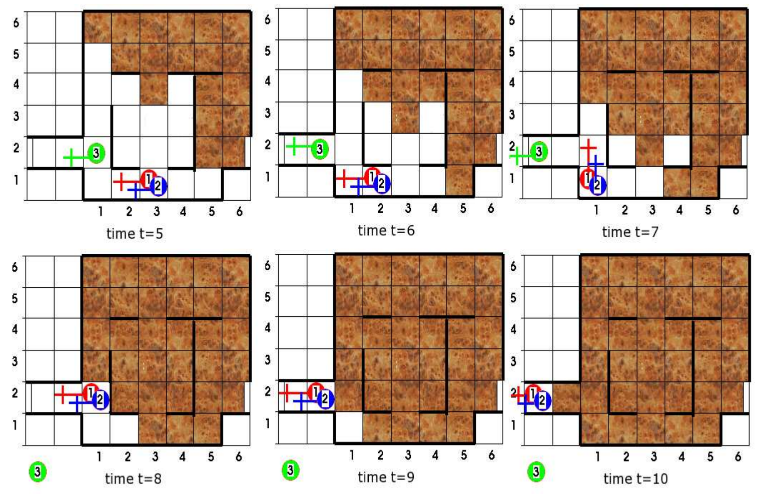
Explanation step. No agents has been confronted to unpredicted events. They have no need for explanation and just trim their hypothesis list.

Communication step. Agents a_1 and a_2 will communicate. Agent a_1 sends its hypothesis (`origin(t=3,at(3,6))`). As this hypothesis is not invalidated by its perception but does not belong to its hypothesis list, a_2 asks for arguments. Agent a_1 sends argument (`fire(t=3,at (3,6))`), and a_2 then computes possible explanations for this and its perception, and gets 6*6*12 possible explanations. Among those, only one contains a common origin for the three observed fires:

```
[rule_propagates_R(t2, from(6, 4)),
rule_propagates_D(t2, from(6, 4)),
rule_propagates_D(t1, from(6, 5)),
rule_propagates_D(t0, from(6, 6)),
rule_propagates_R(t3, from(4, 6)),
rule_propagates_R(t1, from(5, 6)),
rule_propagates_R(t0, from(6, 6)),
origin(t0, at(6, 6))].
```

Agent a_2 then proposes this hypothesis to a_1, which in turn ask for arguments. Finally both agreed upon this hypothesis.

Action step. Agent a_3 continues its escape towards the west exit. Agent a_2 confirms its chosen path with its new hypothesis, and keeps going towards the west door. Agent a_1, however, using its new hypothesis, discover that its escape route is bad. It changes its course to go towards the west exit.



[Round t=5 to 10] From time t=5 to time t=10, agents a_1, a_2 and a_3 exit the building. Agents a_1 and a_2 are closely followed by the fire: one false move would have been fatal! If a_1 did not communicate with a_2 or $_3$ it would not have been able to determine whether the fire was coming from left or right, and would have chosen the east exit and been trapped by the fire.

5 Related Work

Our approach has several facets that can be related to a number of related works. We now introduce some of these related works, starting with the studies of the notion of rumours in social science, that proved to be very inspiring for us.

Rumour in Social Sciences. Rumour is a complex phenomenon that has been the object of numerous studies in social science but is often seen as something that can only bring lies or diffamation. Studies of rumour in social science show, however, that there is more to rumour than just a routing or perception sharing system. Whereas the first studies, done during and after World War II, seem to consider rumour as something dangerous which should be avoided (rumours could lead to moral loss or information leak), more recent stances are somewhat more neutral or positive about it. J.N Kapferer [10] defines rumour as *"the emergence and circulation in the social body of information that either are not yet publicly confirmed by official sources or are denied by them"*. As an unofficial information, it must use alternative ways to be distributed, such as individual communication (gossip, word-of-mouth). He precises that a rumour spreads very quickly because it has value, and because this value decreases over time. Moreover the rightness of the content has no importance. A true rumour spread exactly like a false rumour. The exactitude of the content is not a criteria to define rumour. However, one can choose to take a slightly different perspective on the rumouring process. Shibutani [19] defines rumour as improvised news resulting from a collective discussion process, usually originating from an important and ambiguous event. In his own words, rumour is *"common use of the group individual ressources to get a satisfying intepretation of an event"*. In this case, the rumour is seen as being both an (i) information routing process and (ii) an interpretation and comment adding process. Crucially, the distorsion of information that is often seen as characteristic of rumour is seen as an evolution of the content due to continual interpretation by the group. A crucial aspect of rumour, of course, is that it is a decentralized process. The information propagates without any official control. It is deeply linked with spatial or communication constraint, and can be an efficient way to convey information in spite of these. It is also expected that this process is quite robust to agent error or disparition.

Distributed Diagnosis. The problem of multiagent diagnosis has been studied by Roos and colleagues [15, 16], where a number of distributed entities try to come up with a satisfying global diagnosis of the whole system. They show in particular that the number of messages required to establish this global diagnosis is bound to be prohibitive, unless the communication is enhanced with some suitable protocol. The main difference with our approach lies in the dynamic nature of our context, as well as in the constraints governing agents' interactions that we assume.

Argument-based Interaction. The idea of enhancing communication between agents by adding extra-information that may have the form of arguments has been influential over the last past years in the multiagent community [13]. However, although this approach has several clear advantages (*e.g.* improving expressivity, or facilitating conformance checking), its effectiveness regarding the speed and likelihood of fullfillment of the goal of the interaction has seldom been tested (exceptions are the work of [9], or [11], for instance).

Gossip Problem. Rumours and gossip first appeared within the distributed system community with the *gossip problem*: each agent has a distinct piece of information (called a rumour) to start with. The goal is to make every agent know all the rumours [18]. Some variation of it are the *rumour-spreading problem*, where the agent to communicate to is selected each round by an adversary [1], and the *collect problem.* In the last one, each of n processes in a shared memory system have several pieces of information, and all these processes must learn all the values of all others while making as few as possible primitive read or write operations [17]. It has also been used for reaching consensus [6]. This differs from our approach, mainly because we do not seek to necessarily converge towards a common knowledge of (initially distributed) informations. Also, agents do not modify informations they propagate.

Gossip-based protocols. Each agent has a determined number of neighbours it can communicate with. Each time an agent receives a rumour, it transmits it to to a number of agents chosen at random among its neighbours. Then in turn, each of these agents would do the same. This rumour spreading is analogous to the spreading of an epidemic, which have been the object of mathematical studies [2] and can spread exponentionally fast. Such an information propagation system has first been used for replicated database consistency management [8]. It has been applied to unstructured peer-to-peer communities. Every time an agent detects a change in the system (that would be the rumour), it sends it to a random neighbour, and repeats this operation until it has contacted enough neighbour(s). Some anti-entropy mechanisms are sometimes used to ensure that every agent can get to know each change, even if the rumour has already died out [7]. Another application of these protocols is reliable multicast [3]. It aims at propagating an information from an agent to another agent without a centralised source or knowledge of the system topology, and with a lower cost than with a simple flooding. It is robust to agent deficiency, and very scalable. A variation of it uses weight to enhance the reliability in specific topology [12]. This approach is related to the “recipient selection” aspect of our problem. However, the transmitted information is, again, assumed to be unaffected by agents' reasoning.

Rumour routing. Another approach of rumour as an alternative to flooding is *rumour routing* [4]. In the context of sensor networks, there is a need to transmit queries to agents having observed an event. A fast route between an agent making a request and the agents observing the events might be needed. It can be found by flooding event notifications or queries, and creating a network-wide gradient field [20], but it is a costly approach. Braginsky and Estrin instead propose to use a kind of traceable rumour. Each time an agent observes a new event, it sends an event notification rumour to a random neighbour. This neighbour transmits it in turn to another neighbour, keeping trace of whom it received it from, and how many agent(s) have acted as relay(s), creating rumour paths. When an agent needs to make a query, it sends it to one of its neighbours. If it has heard of the event concerned before, it transmits the query to the agent who told it the rumour, else it transmits to a random neighbour. Eventually, the query will cross the rumour path and be led to the right source. As in the preceding cases, rumour routing propagates pure information, therefore the main studied aspects are the velocity and robustness of these processes.

Reputation Systems. Buchegger and Le Boudec, for instance, use the term of rumour in a reputation system [5]. Their agents can make decisions about the reliability of others agents according to their previously observed behaviour, but also according to what others agents tell about it. In this case, rumour is primarily intended to mean "second-hand information". In this case, agents can keep track of previous partners' behaviours, and also report their observations to other agents. However, these agents are not able to explicitly reason over the justifications governing their decisions.

6 Conclusion

This paper discusses the problem of *efficient propagation of uncertain information* in dynamic environments and critical situations. When a number of (distributed) agents have only partial access to information, the explanation(s) and conclusion(s) they can draw from their observations are inevitably uncertain. In this context, the efficient propagation of information is concerned with two interrelated aspects: spreading the information as quickly as possible, and refining the hypothesis at the same time. We describe a formal framework designed to investigate this class of problem, and propose a simple protocol allowing hypothesis exchange. We also prove some preliminary properties of the protocol and report on an experiment conducted using the described theory.

An obvious advantage of this process (that we observed on the described example) is that agents do not wait to collect all data before providing and propagating hypotheses. In our example this allows agents to escape a building before being caught by the fire. When exactly temporary hypotheses are good enough to be acted upon is to be determined, but this process definitely enable quicker reaction to events than a static centralized data analysis.

The problem is that, of course, it can give incomplete or wrong hypothesis, as the very preliminary analysis of the global properties of the framework suggests. More elaborated communication techniques may then be investigated, allowing agents to backtrack and further refine their hypotheses. In critical situations however, it is unlikely that agents will dispose of sufficient resources to fully synchronize their hypotheses and observations. In consequence, we believe the situations as the one described in our case study to be well suited to such an approach. Further studies are required, however, to determine when exactly this kind of communication would be beneficial, but we expect quickly evolving systems to provide interesting applications. Whereas this paper has mainly focused on agents' reasoning and content selection, we plan to investigate in future research the related problem of recipient selection. Finally, it would also be interesting to consider more complex cases, for instance where agents may have unreliable perceptions of the world, or where malicious propagators of information could adopt an uncooperative behaviour.

Acknowledgments. We would like to thank the anonymous reviewers whose detailed comments helped to greatly improve the paper.

References

1. J. Aspnes and W. Hurwood. Spreading rumors rapidly despite an adversary. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 143–151, 1996.
2. N. Bailey. *The Mathematical Theory of Infectious Diseases*. Charles Griffin and Company, London, 1975.
3. K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2), 1999.
4. D. Braginsky and D. Estrin. Rumor routing algorithm for sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, 2002.
5. S. Buchegger and J. Le Boudec. The effect of rumor spreading in reputation systems for mobile ad-hoc networks. In *Proc. of Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, 2003.
6. B. Chlebus and D. Kowalski. Gossiping to reach consensus. In *Proc., 14th ACM Symp. on Parallel Algorithms and Architectures*, 2002.
7. F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 236–246. IEEE Press, June 2003.
8. A. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 1–12. Vancouver, British Columbia, Canada, 1987.
9. H. Jung and M. Tambe. Argumentation as distributed constraint satisfaction: Applications and results. In *Proceedings of AGENTS01)*, 2001.
10. J.-N. Kapferer. *Rumeurs, le plus vieux média du monde*. Points Actuel, 1990.
11. N. C. Karunatillake and N. R. Jennings. Is it worth arguing? In *Proceedings of First International Workshop on Argumentation in Multi-Agent Systems (ArgMAS 2004)*, pages 62–67, 2004.
12. M.-J. Lin and K. Marzullo. Directional gossip: Gossip in a wide area network. In *European Dependable Computing Conference*, pages 364–379, 1999.
13. S. Parsons, C. Sierra, and N. R. Jennings. Agents that reason and negotiate by arguing. *Journal of Logic and Computation*, 8(3):261–292, 1998.
14. D. Poole. Explanation and prediction: An architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.
15. N. Roos, A. ten Tije, and C. Witteveen. A protocol for multi-agent diagnosis with spatially distributed knowledge. In *Proceedings of AAMAS03*, pages 655–661, 2003.
16. N. Roos, A. ten Tije, and C. Witteveen. Reaching diagnostic agreement in multi-agent diagnosis. In *Proceedings of AAMAS04*, pages 1254–1255, 2004.
17. M. Saks, N. Shavit, and H.Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 351–362, 1991.
18. S.Even and B. Monien. On the number of rounds needed to disseminate information. In *Proc. of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, 1989.
19. T. Shibutani. *Improvised News : A Sociological Study of Rumor*. Indianapolis and New york, 1966.
20. F. Ye, G. Zhong, S. Lu, and L. Zhang. Gradient broadcast: A robust data delivery protocol for large scale sensor networks. *ACM Wireless Networks*, 2005.

A Fibred Tableau Calculus for BDI Logics

Vineet Padmanabhan & Guido Governatori

School of Information Technology & Electrical Engineering
The University of Queensland, Queensland, Australia
`[vnair,guido]@itee.uq.edu.au`

Abstract. In [12, 16] we showed how to combine propositional BDI logics using Gabbay's *fibring* methodology. In this paper we extend the above mentioned works by providing a tableau-based decision procedure for the combined/fibred logics. To achieve this end we first outline with an example two types of tableau systems, (*graph* & *path*), and discuss why both are inadequate in the case of fibring. Having done that we show how to uniformly construct a tableau calculus for the combined logic using Governatori's labelled tableau system **KEM**.

Keywords: Modal and Epistemic Logics for Agent Modelling, Modal Tableaux.

1 Introduction

BDI logics are normal[1] multimodal logics used to formalise the internal mental attitudes of an agent such as beliefs, desires, goals and intentions. Multimodal logics generalise modal logics allowing more than one modal operator to appear in formulae, i.e., a modal operator is named by means of a label, for instance $\Box_i$ which identifies it. Hence a formula like $\Box_i\varphi$ could be interpreted as *φ is believed by the agent i or φ is a goal for agent i etc.* representing respectively the belief and goal of an agent. In addition to the above representation, the traditional BDI logics [17] impose constraints between beliefs, desires and intentions in the form of *interaction axioms* like, $\mathrm{INT}(\varphi) \rightarrow \mathrm{DES}(\varphi)$, $\mathrm{DES}(\varphi) \rightarrow \mathrm{BEL}(\varphi)$, denoting intentions being stronger than desires and desires being stronger than beliefs. Moreover the interaction axioms are *non-homogeneous* in the sense that every modal operator is not restricted to the same system, i.e., the underlying axiom systems for DES is **K** and **D** of modal logic whereas that of BEL is **KD45**. Hence the basic BDI logic $\mathbb{L}$ can be seen as a combination of different component logics plus the two interaction axioms as given below

$$\begin{aligned} \mathbb{L} \equiv\ & (\otimes_{i=1}^{n}\mathbf{KD45}_{\mathrm{BEL}_i}) \otimes (\otimes_{i=1}^{n}\mathbf{KD}_{\mathrm{DES}_i}) \otimes (\otimes_{i=1}^{n}\mathbf{KD}_{\mathrm{INT}_i}) \\ & + \{\mathrm{INT}_i\varphi \rightarrow \mathrm{DES}_i\varphi\} + \{\mathrm{DES}_i\varphi \rightarrow \mathrm{BEL}_i\varphi\} \end{aligned} \tag{1}$$

Any BDI theory, or for that matter any fully-fledged Multi-Agent-System (MAS) theory, modelling rational agents consists of a combined system of logic of beliefs, desires, goals and intentions as mentioned above. They are basically well understood

[1] General modal systems with an arbitrary set of normal modal operators all characterised by the axiom **K**: $\Box(\varphi \rightarrow \psi) \rightarrow (\Box\varphi \rightarrow \Box\psi)$ and the necessitation rule. i.e., $\vdash \varphi / \vdash \Box\varphi$.

standard modal logics *combined together* to model different facets of the agents. A number of researchers have provided such combined systems for different reasons and different applications. However, investigations into a general methodology for combining the different logics involved has been mainly neglected to a large extent. Recently [12, 16] it has been shown that *fibring/dovetailing* [8] can be adopted as a semantic methodology to characterise BDI logics. But in that work they did not provide any decision procedure for the fibred BDI logics. In this paper we extend our previous work so as to provide a tableau decision procedure for the fibred logic which in turn is based on the labelled tableau system **KEM** [10, 9, 1].

The key feature of our tableau system is that it is neither based on resolution nor on standard sequent/tableau techniques. It combines linear tableau expansion rules with natural deduction rules and an analytic version of the cut rule. The tableau rules are supplemented with a powerful and flexible label algebra that allows the system to deal with a large class of intensional logics admitting possible world semantics (non-normal modal logic [11], multi-modal logics [10] and conditional logics [2]). The label algebra is intended to simulate the possible world semantics and it has a very strong relationship with fibring [9].

As far as the field of *combining logics* is concerned, it has been an active research area since some time now and powerful results about the preservation of important properties of the logics being combined has been obtained [13, 5, 4, 20, 21]. Also, investigations related to using fibring as a combining technique in various domains has produced a wealth of results as found in works like [8, 18, 22, 19, 6]. The novelty of combining logics is the aim to develop *general techniques* that allow us to produce combinations of *existing* and well understood logics. Such general techniques are needed for formalising complex systems in a systematic way. Such a methodology can help decompose the problem of designing a complex system into developing components (logics) and combining them.

The advantages of using fibring as a semantic methodology for combining BDI logics as compared to other combining techniques like *fusion* [2] is that the later has the problem of not being able to express interaction axioms, much needed for MAS theories. Fibring is more powerful because of the possibility of adding conditions on the fibring function. These conditions could encode interactions between the two classes of models that are being combined and therefore could represent interaction axioms between the two logics. One such result was shown in [12]. Moreover, fibring does not require the logics to be normal. The drawbacks of other combining techniqiues like *embedding* and *independent combination* when compared to fibring (in the case of BDI logics) has been discussed at length in [15].

The paper is structured as follows. The next section provides a brief introduction to the technique of fibring. Section 3 outlines the path-based and graph-based tableau procedures. Section 4 describes the **KEM** tableau system. The paper concludes with some final remarks.

[2] Normal bimodal and polymodal logics without any interaction axioms are well studied as *fusions* of normal monomodal logics [13, 20].

2 Fibring BDI Logics

Consider the basic BDI logic $\mathbb{L}$ given in (1) which is defined from three component logics, viz., $\mathbf{KD45}_n$ for belief, and $\mathbf{KD}_n$ for desires and intentions. For sake of clarity, consider two of the component logics, $\blacktriangledown_1(\mathbf{KD45})$ and $\blacktriangledown_2(\mathbf{KD})$ and their corresponding languages $\mathscr{L}_{\blacktriangledown_1}, \mathscr{L}_{\blacktriangledown_2}$ built from the respective sets ${}_1$ and ${}_2$ of atoms having classes of models ${}_{\blacktriangledown_1}, {}_{\blacktriangledown_2}$ and satisfaction relations $\models_1$ and $\models_2$. Hence we are dealing with two different systems S_1 and S_2 characterised, respectively, by the class of Kripke models $\mathscr{K}_1$ and $\mathscr{K}_2$. For instance, we know how to evaluate $\Box_1\varphi$ (BEL(φ)) in $\mathscr{K}_1$ (**KD45**) and $\Box_2\varphi$ (DES(φ)) in $\mathscr{K}_2$ (**KD**). We need a method for evaluating $\Box_1$ (resp. $\Box_2$) with respect to $\mathscr{K}_2$ (resp. $\mathscr{K}_1$). In order to do so, we are to link (fibre), via a *fibring* function the model for $\blacktriangledown_1$ with a model for $\blacktriangledown_2$ and build a fibred model of the combination. The fibring function can evaluate (give a yes/no) answer with respect to a modality in S_2, being in S_1 and vice versa. The interpretation of a formula φ of the combined language in the fibred model at a state w can be given as

$$w \models \varphi \text{ if and only if } (w) \models^* \varphi$$

where is a fibring function that maps a world to a model *suitable for interpreting* φ and $\models^*$ is the corresponding satisfaction relation ($\models_1$ for $\blacktriangledown_1$ or $\models_2$ for $\blacktriangledown_2$).

Example 1. Let $\blacktriangledown_1, \blacktriangledown_2$ be two modal logics as given above and let $\varphi = \Box_1\Diamond_2 {}_0$ be a formula on a world w_0 of the fibred semantics. φ belongs to the language $\mathscr{L}_{(1,2)}$ as the outer connective ($\Box_1$) belongs to the language $\mathscr{L}_1$ and the inner connective ($\Diamond_2$) belongs to the language $\mathscr{L}_2$.

By the standard definition we start evaluating $\Box_1$ of $\Box_1\Diamond_2$ at w_0. Hence according to the standard definition we have to check whether $\Diamond_2 {}_0$ is true at every w_1 accessible from w_0 since from the point of view of $\mathscr{L}_1$ this formula has the form $\Box_1 p$ (where $p = \Diamond_2 {}_0$ is atomic). But at w_1 we cannot interpret the operator $\Diamond_2$, because we are in a model of $\blacktriangledown_1$, not of $\blacktriangledown_2$. In order to do this evaluation we need the fibring function which at w_1 points to a world v_0, a world in a model suitable to interpret formulae from $\blacktriangledown_2$. (Fig.1). Now all we have to check is whether $\Diamond_2 {}_0$, is true at v_0 in this last model and this can be done in the usual way. Hence the fibred semantics for the combined language $\mathscr{L}_{(1,2)}$ has models of the form $(\mathscr{F}_1, w_1, v_1, {}_1)$, where $\mathscr{F}_1 = (W_1, \mathsf{R}_1)$ is a frame, and ${}_1$ is the fibring function which associates a model ${}^2_w$ from $\mathscr{L}_2$ with w in $\mathscr{L}_1$ i.e. ${}_1(w) = {}^2_w$.

2.1 Fibring BDI Logics

Let $\mathbf{I}$ be a set of labels representing the modal operators for the intentional states (belief, goal, intention) for a set of agents, and $\blacktriangledown_i, i \in \mathbf{I}$ be modal logics whose respective modalities are $\Box_i, i \in \mathbf{I}$.

Definition 1 *[8] A fibred model is a structure* $(\mathsf{W}, \mathsf{S}, \mathsf{R}, \mathbf{a}, \nu, \tau, \mathbf{F})$ *where*

- W *is a set of possible worlds;*
- S *is a function giving for each w a set of possible worlds,* $\mathsf{S}^w \subseteq \mathsf{W}$*;*

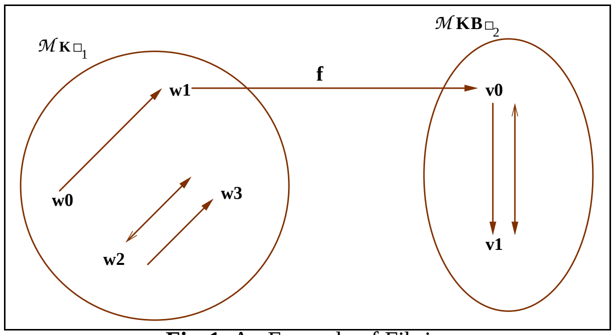


Fig. 1. An Example of Fibring

- R *is a function giving for each* w*, a relation* $\mathsf{R}^w \subseteq \mathsf{S}^w \times \mathsf{S}^w$*;*
- $\mathbf{a}$ *is a function giving the actual world* $\mathbf{a}^w$ *of the model labelled by* w*;*
- v *is an assignment function* $v^w(\ _0) \subseteq \mathsf{S}^w$*, for each atomic* $\ _0$*;*
- τ *is the semantical identifying function* $\tau : \mathsf{W} \to \boldsymbol{I}$*.* $\tau(w) = i$ *means that the model* $(\mathsf{S}^w, \mathsf{R}^w, \mathbf{a}^w, v^w)$ *is a model in* $\mathscr{K}_i$*, we use* W_i *to denote the set of worlds of type* i*;*
- $\mathbf{F}$*, is the set of fibring functions* $\ : \boldsymbol{I} \times \mathsf{W} \mapsto \mathsf{W}$*. A fibring function* $\ $ *is a function giving for each* i *and each* $w \in \mathsf{W}$ *another point (actual world) in* W *as follows:*

$$\ _i(w) = \begin{cases} w & \textit{if } w \in \mathsf{S}\ \textit{ and } \ \in \mathscr{K}_i \\ \textit{a value in } \mathsf{W}_i, & \textit{otherwise} \end{cases}$$

such that if $w \neq w'$ *then* $\ _i(w) \neq \ _i(w')$*. It should be noted that fibring happens when* $\tau(w) \neq i$*. Satisfaction is defined as follows with the usual truth tables for boolean connectives:*

$$w \models \ _0 \textit{ iff } v(w,\ _0) = 1, \textit{ where } \ _0 \textit{ is an atom}$$
$$w \models \Box_i \varphi \textit{ iff } \begin{cases} w \in \ \textit{ and } \ \in \mathscr{K}_i \textit{ and } \forall w'(w\mathsf{R}w' \to w' \models \varphi), \textit{or} \\ w \in \ , \textit{ and } \ \notin \mathscr{K}_i \textit{ and } \forall \ \in \mathbf{F},\ _i(w) \models \Box_i \varphi. \end{cases}$$

We say the model satisfies φ *iff* $w_0 \models \varphi$*.*

A fibred model for $\blacktriangledown_{\mathbf{I}}$ can be generated from fibring the semantics for the modal logics $\blacktriangledown_i, i \in \mathbf{I}$. The detailed construction is given in [16]. Also, to accommodate the interaction axioms specific constraints need to be given on the fibring function. In [12] we outline the specific conditions required on the fibring function to accommodate axiom schemas of the type $G^{a,b,c,d}$:[3]. We do not want to get into the details here as the main theme of this paper is with regard to tableaux decision procedures for fibred logics.

What we want to point out here, however, is that the fibring construction given in [12, 16] works for normal (multi-)modal logics as well as non-normal modal logics.

3 Multimodal Tableaux

In the previous sections we showed that BDI logics are normal multimodal logics with a set of interaction axioms and introduced general techniques like fibring to explain

[3] $G^{a,b,c,d} \Diamond_a \Box_b \varphi \to \Box_c \Diamond_d \varphi$.

such combined systems. In this section, before getting into the details related to the constructs needed for a tableau calculus for a fibred/combined logic, we outline with an example two types of tableau systems (*graph* & *path*) that can be used to reason about the knowledge/beliefs of BDI agents in a multi-agent setting. We discuss why both types are inadequate in the case of fibring. Having done that, in the next section, we describe how to uniformly construct a sound and complete tableau calculus for the combined logic from calculi for the component logics.

Example 2. (The Friends Puzzle) [3] Consider the agents Peter, John and Wendy with modalities $\Box_p, \Box_j$, and $\Box_w$. John and Peter have an `appointment`. Suppose that Peter knows the `time` of appointment. Peter knows that John knows the `place` of their appointment. Wendy knows that if Peter knows the `time` of appointment, then John knows that too (since John and Peter are friends). Peter knows that if John knows the `place` and the `time` of their appointment, then John knows that he has an `appointment`. Peter and John satisfy the axioms T and 4. Also, if Wendy knows something then Peter knows the same thing (suppose Wendy is Peter's wife) and if Peter knows that John knows something then John knows that Peter knows the same thing.

The Knowledge/belief base for Example 2 can be formally given as follows;

1. $\Box_p time$	A_1 $T_p : \Box_p\varphi \rightarrow \varphi$
2. $\Box_p\Box_j place$	A_2 $4_p : \Box_p\varphi \rightarrow \Box_p\Box_p\varphi$
3. $\Box_w(\Box_p time \rightarrow \Box_j time)$	A_3 $T_j : \Box_j\varphi \rightarrow \varphi$
4. $\Box_p\Box_j(place \wedge time \rightarrow appointment)$	A_4 $4_j : \Box_j\varphi \rightarrow \Box_j\Box_j\varphi$
	A_5 $I_{wp} : \Box_w\varphi \rightarrow \Box_p\varphi$
	A_6 $S_{pj} : \Box_p\Box_j\varphi \rightarrow \Box_j\Box_p\varphi$

Fig. 2. Knowledge base related to the Friend's puzzle.

So we have a modal language consisting of three modalities $\Box_p, \Box_j$ and $\Box_w$ denoting respectively the agents Peter, John and Wendy and characterised by the set $A = \{A_i \mid i = 1, \dots, 6\}$ of interaction axioms. Suppose now that one wants to show that each of the friends knows that the other one knows that he has an appointment, i.e, one wants to prove

$$\Box_j\Box_p appointment \wedge \Box_p\Box_j appointment \tag{2}$$

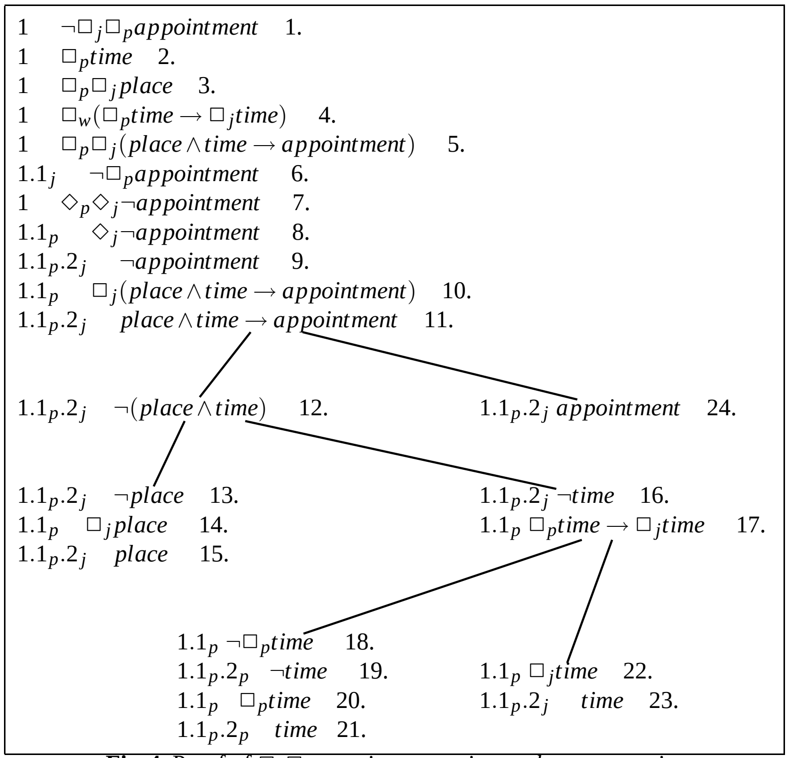
is a theorem of the knowledge-base. The tableaux rules for a logic corresponding to the Friends puzzle are given in Fig.3 [14], and the tableaux proof for (2) is given in Fig.4 [14]. The tableaux in Fig.4. is a prefixed tableau [7] where the accessibility relations are encoded in the structure of the name of the worlds. Such a representation is often termed as a *path* representation. We show the proof of the first conjunct and the proof runs as follows. Item 1 is the negation of the formula to be proved; 2, 3, 4 and 5 are from Example 2; 6 is from 1 by a $\Diamond$-rule; 7 is from 6 by an S_{pj}-rule; 8 is from 7 by a $\Diamond$-rule; 9 is from 8 by a $\Diamond$-rule; 10 is from 5 by a $\Box$-rule; 11 is from 10 by a $\Box$-rule. 12 and 24 are from 11 by a $\vee$-rule; 13 and 16 are from 12 by a $\vee$-rule; 14 is from 3 by a $\Box$-rule; 15 is from 14 by a $\Box$-rule; the branch closes by 13 and 15; 17 is from 4 by an I_{wp}-rule; 18 and 22 are from 17 by a $\vee$-rule; 19 is from 18 by a $\Diamond$-rule; 20 is from 2 by a 4_p-rule; 21 is from 20 by a $\Box$-rule; the branch closes by 19 and 21; 23 is from 22 by a $\Box$-rule; the branch closes by 16 and 23; by 9 and 24 the remaining branch too closes.

$\wedge$-rules	$\dfrac{\sigma\ \varphi\wedge\psi}{\begin{array}{c}\sigma\ \varphi\\ \sigma\ \psi\end{array}}$	$\dfrac{\sigma\ \neg(\varphi\vee\psi)}{\begin{array}{c}\sigma\ \neg\varphi\\ \sigma\ \neg\psi\end{array}}$	$\dfrac{\sigma\ \neg(\varphi\rightarrow\psi)}{\begin{array}{c}\sigma\ \varphi\\ \sigma\ \psi\end{array}}$		For any prefix σ
$\vee$-rules	$\dfrac{\sigma\ \varphi\vee\psi}{\sigma\ \varphi \mid \sigma\ \psi}$	$\dfrac{\sigma\ \neg(\varphi\wedge\psi)}{\sigma\ \neg\varphi \mid \sigma\ \neg\psi}$	$\dfrac{\sigma\ \varphi\rightarrow\psi}{\sigma\ \neg\varphi \mid \sigma\ \neg\psi}$		For any prefix σ
$\neg\neg$-rules	$\dfrac{\sigma\neg\neg\varphi}{\sigma\varphi}$				For any prefix σ
$\Diamond$-rules	$\dfrac{\sigma\ \Diamond_i\varphi}{\sigma.n_i\ \varphi}$	$\dfrac{\sigma\ \neg\Box_i\varphi}{\sigma.n_i\ \neg\varphi}$			if the prefix $\sigma.n_i$ is new to the branch ($i\in\{1,\ldots,m\}$)
$\Box$-rules	$\dfrac{\sigma\ \Box_i\varphi}{\sigma.n_i\ \varphi}$	$\dfrac{\sigma\ \neg\Diamond_i\varphi}{\sigma.n_i\ \neg\varphi}$			If the prefix $\sigma.n_i$ already occurs on the branch ($i\in\{1,\ldots,m\}$)
$T_p$rules:	$\dfrac{\sigma\ \Box_p\varphi}{\sigma\ \varphi}$	$\dfrac{\sigma\ \neg\Diamond_p\varphi}{\sigma\ \neg\varphi}$	$\dfrac{\sigma\ \varphi}{\sigma\ \Diamond_p\varphi}$		
$T_j$rules:	$\dfrac{\sigma\ \Box_j\varphi}{\sigma\ \varphi}$	$\dfrac{\sigma\ \neg\Diamond_j\varphi}{\sigma\ \neg\varphi}$	$\dfrac{\sigma\ \varphi}{\sigma\ \Diamond_j\varphi}$		
4_prules:	$\dfrac{\sigma\ \Box_p\varphi}{\sigma.n_p^*\Box_p\varphi}$	$\dfrac{\sigma\ \neg\Diamond_p\varphi}{\sigma.n_p^*\Box_p\neg\varphi}$	$\dfrac{\sigma.n_p\ \Diamond_p\varphi}{\sigma\ \Diamond_p\varphi}$	$\dfrac{\sigma.n_p\ \neg\Box_p\varphi}{\sigma\ \Diamond_p\neg\varphi}$	
4_jrules:	$\dfrac{\sigma\ \Box_j\varphi}{\sigma.n_j^*\Box_j\varphi}$	$\dfrac{\sigma\ \neg\Diamond_j\varphi}{\sigma.n_j^*\Box_j\neg\varphi}$	$\dfrac{\sigma.n_j\ \Diamond_j\varphi}{\sigma\ \Diamond_j\neg\varphi}$	$\dfrac{\sigma.n_j\ \neg\Box_j\varphi}{\sigma\ \Diamond_j\neg\varphi}$	
I_{wp}rules:	$\dfrac{\sigma\ \Box_w\varphi}{\sigma.n_p^*\varphi}$	$\dfrac{\sigma\ \neg\Diamond_w\varphi}{\sigma.n_p^*\neg\varphi}$	$\dfrac{\sigma.n_p\ \varphi}{\sigma\ \Diamond_w\varphi}$		
S_{pj}rules:	$\dfrac{\sigma\ \Box_p\Box_j\varphi}{\sigma.n_j^*\Box_p\varphi}$	$\dfrac{\sigma\ \neg\Diamond_p\Diamond_j\varphi}{\sigma.n_j^*\Box_p\neg\varphi}$	$\dfrac{\sigma.n_j\ \Diamond_p\varphi}{\sigma\ \Diamond_p\Diamond_j\varphi}$	$\dfrac{\sigma.n_j\ \neg\Box_p\varphi}{\sigma\ \Diamond_p\Diamond_j\neg\varphi}$	

$(*)$ prefix already occurs on the branch

Fig. 3. Tableau rules corresponding to the Friend's Puzzle.

In a similar manner the tableaux proof for (2) using a *graph* representation where the accessibility relations are represented by means of an explicit and separate graph of named nodes is given in Fig.6. Each node is associated with a set of prefixed formulae and choice allows any inclusion axiom to be interpreted as a *rewriting rule* into the path structure of the graph. The proof uses the rules given in Fig.5. which is often referred to as the Smullyan-Fitting uniform notation. We will be using this notation in the next section for our **KEM** tableaux system. The proof for (2) as given in [3] runs as follows. Steps 1-4 are from Fig.2 and 5 is the first conjunct of (2). Using π-rule we get items 6 and 7 (from 5) and 8 and 9 (from 6). We get 10 from 7 using axiom A_6 in Fig.2 and ρ-rule in Fig.5. Similarly 11 is from 9 via A_6 and ρ-rule. By making use of the ν-rule in Fig.5 we get 12 (from 4 and 10) and 13 (from 12 and 11). 14a and 14b are from 13 using β-rule ("a" and "b" denote the two branches created by the application of β-rule). Branch "a" (14a) closes with 8. Applying β-rule again we get 15ba and 15bb from 14b ("ba" and "bb" denote the two branches created by the application of β-rule). Applying ν-rule we get 16ba (from 3 and 10) and 17ba (from 16ba and 11). Branch "ba" closes because of 15ba and 17ba. We get 16bb from 10 via axiom A_5 in Fig.2 and π-rule in Fig.5. Similarly from 2 and 16bb by using ν-rule we get 17bb. We get 18bba and 18bbb



1 $\neg\Box_j\Box_p appointment$ 1.
1 $\Box_p time$ 2.
1 $\Box_p\Box_j place$ 3.
1 $\Box_w(\Box_p time \to \Box_j time)$ 4.
1 $\Box_p\Box_j(place \wedge time \to appointment)$ 5.
1.1_j $\neg\Box_p appointment$ 6.
1 $\Diamond_p\Diamond_j\neg appointment$ 7.
1.1_p $\Diamond_j\neg appointment$ 8.
$1.1_p.2_j$ $\neg appointment$ 9.
1.1_p $\Box_j(place \wedge time \to appointment)$ 10.
$1.1_p.2_j$ $place \wedge time \to appointment$ 11.
$1.1_p.2_j$ $\neg(place \wedge time)$ 12.
$1.1_p.2_j$ $appointment$ 24.
$1.1_p.2_j$ $\neg place$ 13.
1.1_p $\Box_j place$ 14.
$1.1_p.2_j$ $place$ 15.
$1.1_p.2_j$ $\neg time$ 16.
1.1_p $\Box_p time \to \Box_j time$ 17.
1.1_p $\neg\Box_p time$ 18.
$1.1_p.2_p$ $\neg time$ 19.
1.1_p $\Box_p time$ 20.
$1.1_p.2_p$ $time$ 21.
1.1_p $\Box_j time$ 22.
$1.1_p.2_j$ $time$ 23.

Fig. 4. Proof of $\Box_j\Box_p$ appointment using *path* representation

from 17bb by applying the β-rule ("bba" and "bbb" denote the branches created by the β-rule). By using ν-rule we get 19bba (from 18bba and 11). Branch "bba" (19bba) closes with 15bb. From 18bbb using π-rule we get 19bbb and 20bbb. From 10 and 20bbb via axiom A_2 (in Fig.2) and ρ-rule (in Fig.5) we get 21bbb. By applying ν-rule to 1 and 21bbb we get 22bbb as a result of which the branch "bbb" closes (22bbb and 19bbb).

It should be noted that axiom schemas like $A_1,\ldots,A_6$ of Example 2 given in Fig. 2 belong to the class of axioms called *inclusion axioms*. In particular they belong to axiom sets of the form, $\Box_{i_1}\ldots\Box_{i_n} \to \Box_{i'_1}\ldots\Box_{i'_m}$ $(i_n > 0, i'_m \geq 0)$, which in turn characterise the class of *normal modal logics* called *inclusion modal logics*. As shown in [3], for each axiom schema of the above type the corresponding *inclusion* property on the *accessibility relation* can be given as

$$\mathsf{R}_{i_1} \circ \mathsf{R}_{i_2} \circ \ldots \mathsf{R}_{i_n} \supseteq \mathsf{R}_{i'_1} \circ \mathsf{R}_{i'_2} \ldots \circ \mathsf{R}_{i'_m} \tag{3}$$

where "$\circ$" denotes the relation composition $\mathsf{R}_{i_1} \circ \mathsf{R}_{i_2} = \{(w,w'') \in W \times W \mid \exists w' \in W$ such that $(w,w') \in \mathsf{R}_{i_1}$ and $(w',w'') \in \mathsf{R}_{i_2}\}$. This inclusion property is used to rewrite items 7. $(w_0\mathsf{R}_{john}w_1)$ and 9. $(w_1\mathsf{R}_{peter}w_2)$ of the proof given in Fig.6 so as to derive a new path $(w_0\mathsf{R}_{peter}w_3)$ and $(w_3\mathsf{R}_{john}w_2)$ as in items 10. and 11. The corresponding tableaux rule for this property is given as ρ-rule (5) in Fig.5. Also, the type of interaction axiom schemas of Example 2 involves the interaction between the *same mental attitude* of *different agents*. There is also another type where there is interaction between

(1) $\dfrac{w:\alpha}{\begin{array}{c}w:\alpha_1\\ w:\alpha_2\end{array}}\,\alpha$-rule

(2) $\dfrac{w:\beta}{w:\beta_1 \mid w:\beta_2}\,\beta$-rule

(3) $\dfrac{w:\nu_i \quad w\rho_i w'}{w':\nu_i^0}\,\nu$-rule where $w\rho_i w'$ is *available* on the branch

(4) $\dfrac{w:\pi_i}{\dfrac{w':\pi_i^0}{w:\rho_i w'}}\,\pi$-rule where w' is *new* on the branch

(5) $\dfrac{w\rho_{s_1}w_1 \dots w_{m-1}\rho_{s_m}w'}{\begin{array}{c}w\rho_{i_1}w'_1\\ \vdots\\ w'_{n-1}\rho_{i_n}w'\end{array}}\,\rho$-rule where $w'_1, \dots, w'_{n-1}$ are *new* on the branch and $\Box_{i_1} \dots \Box_{i_n}\varphi \to \Box_{i'_1} \dots \Box_{i'_m}\varphi \in \mathrm{A}$

α	α_1	α_2
T $(\varphi \wedge \psi)$	**T** φ	**T** ψ
F $(\varphi \vee \psi)$	**F** φ	**F** ψ
F $(\varphi \to \psi)$	**T** φ	**F** ψ
F $(\neg\varphi)$	**T** φ	**T** φ

(a) $\wedge$-formulae

β	β_1	β_2
F $(\varphi \wedge \psi)$	**F** φ	**F** ψ
T $(\varphi \vee \psi)$	**T** φ	**T** ψ
T $(\varphi \to \psi)$	**F** φ	**T** ψ
T $(\neg\varphi)$	**F** φ	**F** φ

(b) $\vee$-formulae

ν_i	ν_0
T$\Box_i\varphi$	**T**φ
F$\Diamond_i\varphi$	**F** φ

(c) $\Box$-formulae

π_i	π_0
F$\Box_i\varphi$	**F**φ
T$\Diamond_i\varphi$	**T** φ

(d) $\Diamond$-formulae

Fig. 5. Tableaux rules based on uniform notation for propositional inclusion modal logics. [3].

different mental attitudes of the *same agent*. The BDI interaction axioms given in (1) is of the later type. In the coming sections we will show that the **KEM** tableau can deal with both types of interaction axioms.

As pointed out in [3], the main difference between the two types of tableaux, (graph and path), is in the use of ν-rule. In the case of *path* representation one needs to use a specific ν-rule for each logic as can be seen from Fig.3. These rules code the properties of the accessibility relations so as to express complex relations between prefixes depending on the logic. Whereas in the case of *graph* representation the accessibility relations are given explicitly. Also, it has been pointed out in [3] that the approach based on path representation can be used only for some subclasses of inclusion axioms and therefore difficult to extend the approach to the whole class of multi-modal systems.

4 Labelled Tableau for Fibred BDI Logic

In this section we show how to adapt **KEM**, a labelled modal tableaux system, to deal with the fibred combination of BDI logics. In labelled tableaux systems, the object language is supplemented by labels meant to represent semantic structures (possible worlds in the case of modal logics). Thus the formulas of a labelled tableaux system are expressions of the form $A : i$, where A is a formula of the logic and i is a label. The intuitive interpretation of $A : i$ is that A is true at (the possible world(s) denoted by) i.

1.	$w_0 : \mathbf{T}\Box_p time$	14b.	$w_2 : \mathbf{F}(\text{place} \wedge \text{time})$
2.	$w_0 : \mathbf{T}\Box_w(\Box_p time \rightarrow \Box_j time)$	15ba.	$w_2 : \mathbf{F}\ \text{place}$
3.	$w_0 : \mathbf{T}\Box_p\Box_j place$	16ba.	$w_3 : \mathbf{T}\Box_j place$
4.	$w_0 : \mathbf{T}\Box_p\Box_j(place \wedge time \rightarrow appointment)$	17ba.	$w_2 : \mathbf{T}\text{place}$
5.	$w_0 : \mathbf{F}\Box_j\Box_p appointment$		$\times$
6.	$w_1 : \mathbf{F}\Box_p appointment$	15bb.	$w_2 : \mathbf{F}time$
7.	$w_0 \mathsf{R}_{john} w_1$	16bb.	$w \mathsf{R}_{wife} w_3$
8.	$w_2 : \mathbf{F}\ appointment$	17bb.	$w_3 : \mathbf{T}(\Box_p time \rightarrow \Box_j time)$
9.	$w_1 \mathsf{R}_{peter} w_2$	18bba.	$w_3 : \mathbf{T}\ \Box_j time$
10.	$w_0 \mathsf{R}_{peter} w_3$	19bba.	$w_2 : \mathbf{T}time$
11.	$w_3 \mathsf{R}_{john} w_2$		$\times$
12.	$w_3 : \mathbf{T}\Box_j(place \wedge time \rightarrow appointment)$	18bbb.	$w_3 : \mathbf{F}\Box_p time$
13.	$w_2 : \mathbf{T}\ (place \wedge time \rightarrow appointment)$	19bbb.	$w_4 : \mathbf{F}time$
14a.	$w_2 : \mathbf{T}\ appointment$	20bbb	$w_3 \mathsf{R}_{peter} w_4$
	$\times$	21bbb.	$w_0 \mathsf{R}_{peter} w_4$
		22bbb.	$w_4 : \mathbf{T}\ time$
			$\times$

Fig. 6. Proof of $\Box_j\Box_p$ using *graph* representation.

KEM's inferential engine is based on a combination of standard tableaux linear expansion rules and natural deduction rules supplemented by an analytic version of the cut rule. In addition it utilises a sophisticated but powerful label formalism that enables the logic to deal with a large class of modal and non-classical logics. Furthermore the label mechanism corresponds to fibring and thus it is possible to define tableaux systems for multi-modal logic by a seamless combination of the (sub)tableaux systems for the component logics of the combination.

It is not possible in this paper to give a full presentation of **KEM** for fully fledged BDI logic supplemented with the interaction axioms given in Example 2. (for a comprehensive presentation see [9]). Accordingly we will limit ourselves to a single modal operator for each agent and we will show how to characterise the axioms and the interaction of example 2.

4.1 Label Formalism

KEM uses *Labelled Formulas* (*L*-formulas for short), where an *L*-formula is an expression of the form $A : i$, where A is a wff of the logic, and i is a label. For fibred BDI logic (from now on **FBL**) we need to have labels for various modalities (belief, desire, intention) for each agent. However, as we have just explained we will consider only one modality and thus will have only labels for the agents.

The set of atomic labels, $\Im_1$, is then given as

$$\Im_1 = \bigcup_{i \in Agt} \Phi^i,$$

where Agt is the set of agents. Every Φ^i is partitioned into (non-empty) sets of variables and constants: $\Phi^i = \Phi^i_V \cup \Phi^i_C$ were $\Phi^i_V = \{W^i_1, W^i_2, \dots\}$ and $\Phi^i_C = \{w^i_1, w^i_2, \dots\}$. Φ_C and Φ_V denote the set of constants and the set of variables. We also add a set of auxiliary un indexed atomic labels $\Phi^A = \Phi^A_V = \{W_1, W_2, \dots\} \cup \Phi^A_C = \{w_1, w_2, \dots\}$, that will be used in unifications and proofs.

Definition 1 (labels) *A label $u \in \Im$ is either (i) an element of the set Φ_C, or (ii) an element of the set Φ_V, or (iii) a path term (u', u) where (iiia) $u' \in \Phi_C \cup \Phi_V$ and (iiib) $u \in \Phi_C$ or $u = (v', v)$ where (v', v) is a label.*

As an intuitive explanation, we may think of a label $u \in \Phi_C$ as denoting a world (a *given* one), and a label $u \in \Phi_V$ as denoting a set of worlds (*any* world) in some Kripke model. A label $u = (v', v)$ may be viewed as representing a path from v to a (set of) world(s) v' accessible from v (the world(s) denoted by v).

For any label $u = (v', v)$ we shall call v' the *head* of u, v the *body* of u, and denote them by $h(u)$ and $b(u)$ respectively. Notice that these notions are recursive (they correspond to projection functions): if $b(u)$ denotes the body of u, then $b(b(u))$ will denote the body of $b(u)$, and so on. We call each of $b(u)$, $b(b(u))$, etc., a *segment* of u. The length of a label u, $\ell(u)$, is the number of atomic labels in it. $s^n(u)$ will denote the segment of u of length n and we shall use $h^n(u)$ as an abbreviation for $h(s^n(u))$. Notice that $h(u) = h^{\ell(u)}(u)$. Let u be a label and u' an atomic label. We use $(u'; u)$ as a notation for the label (u', u) if $u' \neq h(u)$, or for u otherwise. For any label $u, \ell(u) > n$, we define the *counter-segment-n* of u, as follows (for $n < k < \ell(u)$):

$$c^n(u) = h(u) \times (\cdots \times (h^k(u) \times (\cdots \times (h^{n+1}(u), w_0))))$$

where w_0 is a dummy label, i.e., a label not appearing in u (the context in which such a notion occurs will tell us what w_0 stands for). The counter-segment-n defines what remains of a given label after having identified the segment of length n with a ‘dummy’ label w_0. The appropriate dummy label will be specified in the applications where such a notion is used. However, it can be viewed also as an independent atomic label. In the context of fibring w_0 can be thought of as denoting the actual world obtained via the fibring function from the world denoted by $s^{n(u)}$.

So far we have provided definitions about the structure of the labels without regard to the elements they are made of. The following definitions will be concerned with the type of world symbols occurring in a label.

We say that a label u is *i-preferred* iff $h(u) \in \Phi^i$; a label u is *i-pure* iff each segment of u of length $n > 1$ is i-preferred.

4.2 Label Unifications

The basic mechanism of **KEM** is its logic dependent label unification. In the same way as each modal logic is characterised by a combination of modal axioms (or semantic conditions on the model), **KEM** defines a unification for each modality and axiom/semantic condition and then combines them in a recursive and modular way. In particular we use what we call unification to determine whether the denotation of two labels have a non empty intersection, or in other terms whether two labels can be mapped to the same possible world in the possible worlds semantics.

The second key issue is the ability to split labels and to work with parts of labels. The mechanism permits the encapsulation of operations on sub-labels. This is an important feature that, in the present context, allows us to correlate unifications and fibring functions. Given the modularity of the approach the first step of the construction is to define unifications (pattern matching for labels) corresponding to the single modality in the logic we want to study.

Every unification is built from a basic unification defined in terms of a substitution $\rho : \Im_1 \mapsto \Im$ such that:

$$\begin{aligned}\rho : &\mathbf{1}_{\Phi_C}\\ &\Phi_V^i \mapsto \Im^i \text{ for every } i \in Agt\\ &\Phi_V^A \mapsto \Im\end{aligned}$$

Accordingly we have that two atomic ("world") labels u and v σ-unify iff there is a substitution ρ such that $\rho(u) = \rho(v)$. We shall use $[u;v]\sigma$ both to indicate that there is a substitution ρ for u and v, and the result of the substitution. The σ-unification is extended to the case of composite labels (path labels) as follows:

$$\begin{aligned}[i;j]\sigma = k \text{ iff } \; &\exists\rho : h(k) = \rho(h(i)) = \rho(h(j)) \text{ and}\\ &b(k) = [b(i);b(j)]\sigma\end{aligned}$$

Clearly σ is symmetric, i.e., $[u;v]\sigma$ iff $[v;u]\sigma$. Moreover this definition offers a flexible and powerful mechanism: it allows for an independent computation of the elements of the result of the unification, and variables can be freely renamed without affecting the result of a unification.

We are now ready to introduce the unifications corresponding to the modal operators at hand, i.e., $\Box_w$, $\Box_j$ and $\Box_p$. We can capture the relationship between $\Box_w$ and $\Box_p$ by extending the substitution ρ by allowing a variable of type w to be mapped to labels of the same type and of type p.

$$\rho^w(W^w) \in \Im^w \cup \Im^p$$

Then the unification σ^w is obtained from the basic unification σ by replacing ρ with the extended substitution ρ^w. This procedure must be applied to all pairs of modalities $\Box_1, \Box_2$ related by the interaction axiom $\Box_1\varphi \to \Box_2\varphi$.

For the unifications for $\Box_p$ and $\Box_j$ (σ^p and σ^j) we assume that the labels involved are i-pure. First we notice that these two modal operators are **S4** modalities thus we have to use the unification for this logic.

$$[u;v]\sigma^{S4} = \begin{cases}[u;v]\sigma^D \text{ if } \ell(u) = \ell(v)\\ [u;v]\sigma^T \text{ if } \ell(u) < \ell(v), h(u) \in \Phi_C\\ [u;v]\sigma^4 \text{ if } \ell(u) < \ell(v), h(u) \in \Phi_V\end{cases} \tag{4}$$

It is worth noting that the conditions on axiom unifications are needed in order to provide a deterministic unification procedure. The σ^T and σ^4 are defined as follows:

$$[u;v]\sigma^T = \begin{cases}[s^{\ell(v)}(u);v]\sigma \text{ if } \ell(u) > \ell(v), \text{ and}\\ \qquad\qquad \forall n \geq \ell(v), [h^n(u);h(v))]\sigma = [h(u);h(v)]\sigma\\ [u;s^{\ell(u)}(v)]\sigma \text{ if } \ell(u) > \ell(v), \text{ and}\\ \qquad\qquad \forall n \geq \ell(u), [h(u);h^n(v)]\sigma = [h(u);h(v)]\sigma\end{cases}$$

The above unification allows us to unify to labels such that the segment of the longest with the length of the other label and the other label unify, provided that all remaining elements of the longest have a common unification with the head of the shortest. This means that after a given point the head of the shortest is always included in its extension, and thus it is accessible from itself, and consequently we have reflexivity.

$$[u;v]\sigma^4 = \begin{cases} c^{\ell(u)}(v) \text{ if } \ell(v) > \ell(u), h(u) \in \Phi_V \text{ and} \\ \qquad w_0 = [u; s^{\ell(u)}(v)]\sigma \\ c^{\ell(v)}(u) \text{ if } \ell(u) > \ell(v), h(v) \in \Phi_V \text{ and} \\ \qquad w_0 = [s^{\ell(v)}(u); v]\sigma \end{cases}$$

In this case we have that the shortest label unifies with the segment with the same length of the longest and that the head of the shortest is variable. A variable stands for all worlds accessible from the predecessor of it. Thus, given transitivity every element extending the segment with length of the shortest is accessible from this point.

Then a unification corresponding to axiom A6 from Example is 2.

$$[u;v]\sigma^{S_{p,j}} = \begin{cases} c^{m+n}(v) \text{ if } h(u) \in \Phi_V^j \text{ and } c^n(v) \text{ is } p\text{-pure, and} \\ \qquad h^{\ell(u)-1}(u) \in \Phi_V^p \text{ and } c^n(v) \text{ is } j\text{-pure, and} \\ \qquad w_0 = [s^{\ell(u)-2}(u); s^m(v)]\sigma \\ c^{m+n}(u) \text{ if } h(v) \in \Phi_V^j \text{ and } c^n(u) \text{ is } p\text{-pure, and} \\ \qquad h^{\ell(v)-1}(v) \in \Phi_V^p \text{ and } c^n(u) \text{ is } j\text{-pure and} \\ \qquad w_0 = [s^m(u); s^{\ell(v)-2}(v)]\sigma \end{cases}$$

This unification allows us to unify two labels such that in one we have a sequence of a variable of type p followed by a variable of type j and a label where we have a sequence of labels of type j followed by a sequence of labels of type p.

The unification for $\Box_p$ and $\Box_j$ are just the combination of the three unifications given above. Finally the unification for the logic **L** defined by the axioms A1–A6 is obtained from the following recursive unification

$$[u;v]\sigma_L = \begin{cases} [u;v]\sigma^{w,p,j} \\ [c^m(u); c^n(v)]\sigma^{w,p,j} \text{ where } w_0 = [s^m(u); s^n(v)]\sigma_L \end{cases}$$

$\sigma^{w,p,j}$ is the simple combination of the unifications for the three modal operators. Having accounted for the unification we now give the inference rules used in **KEM** proofs.

4.3 Inference Rules

For the inference rules we use the Smullyan-Fitting unifying notation [7].

$$\frac{\alpha : u}{\begin{array}{c}\alpha_1 : u \\ \alpha_2 : u\end{array}} (\alpha) \qquad \frac{\begin{array}{c}\beta : u \\ \beta_i^c : v\end{array} (i = 1,2)}{\beta_{3-i} : [u;v]\sigma} (\beta)$$

The α-rules are just the familiar linear branch-expansion rules of the tableau method. The β-rules are nothing but natural inference patterns such as Modus Ponens, Modus

Tollens and Disjunctive syllogism generalised to the modal case. In order to apply such rules it is required that the labels of the premises unify and the label of the conclusion is the result of their unification.

$$\frac{\nu^i : u}{\nu^i_0 : (W^i_n, u)}(\nu) \qquad\qquad \frac{\pi^i : u}{\pi^i_0 : (w^i_n, u)}(\pi)$$

where W^i_n is a new label.

The ν and π rules are the normal expansion rule for modal operators of labelled tableaux with free variable. The intuition for the ν rule is that if $\Box_i A$ is true at u, then A is true at all worlds accessible via R_i from u, and this is the interpretation of the label (W^i_n, u); similarly if $\Box_i A$ is false at u (i.e., $\neg\mathbf{B}A$ is true), then there must be a world, let us say w^i_n accessible from u, where $\neg A$ is true. A similar intuition holds when u is not i-preferred, but the only difference is that we have to make use of the fibring function instead of the accessibility relation

$$\frac{}{A : u \quad | \quad \neg A : u} \qquad (PB)$$

The "Principle of Bivalence" represents the semantic counterpart of the cut rule of the sequent calculus (intuitive meaning: a formula A is either true or false in any given world). PB is a zero-premise inference rule, so in its unrestricted version can be applied whenever we like. However, we impose a restriction on its application. PB can be only applied w.r.t. immediate sub-formulas of unanalysed β-formulas, that is β formulas for which we have no immediate sub-formulas with the appropriate labels in the tree.

$$\begin{array}{l} A : u \\ \dfrac{\neg A : v}{\times}[\text{ if } [u; v]\sigma] \end{array} \qquad (PNC)$$

The *Principle of Non-Contradiction* (PNC) states that two labelled formulas are $\sigma_{\mathbf{L}}$-complementary when the two formulas are complementary and their labels $\sigma_{\mathbf{L}}$-unify.

4.4 Proof Search

Let $\Gamma = \{X_1, \ldots, X_m\}$ be a set of formulas. Then $\mathscr{T}$ is a **KEM**-*tree for* Γ if there exists a finite sequence $(\mathscr{T}_1, \mathscr{T}_2, \ldots, \mathscr{T}_n)$ such that (i) $\mathscr{T}_1$ is a 1-branch tree consisting of $\{X_1 : t_1, \ldots, X_m : t_m\}$; (ii) $\mathscr{T}_n = \mathscr{T}$, and (iii) for each $i < n$, $\mathscr{T}_{i+1}$ results from $\mathscr{T}_i$ by an application of a rule of **KEM**. A branch θ of a **KEM**-tree $\mathscr{T}$ of L-formulas is said to be $\sigma_{\mathbf{L}}$-*closed* if it ends with an application of *PNC*, open otherwise. As usual with tableau methods, a set Γ of formulas is checked for consistency by constructing a **KEM**-tree for Γ. Moreover we say that a formula A is a **KEM**-*consequence of a set of formulas* $\Gamma = \{X_1, \ldots, X_n\}$ ($\Gamma \vdash_{\mathbf{KEM}(L)} A$) if a **KEM**-tree for $\{X_1 : u_1, \ldots, X_n : u_n, \neg A : v\}$ is closed using the unification for the logic L, where $v \in \Phi^A_C$, and $u_i \in \Phi^A_V$. The intuition behind this definition is that A is a consequence of Γ when we take Γ as a set of global assumptions [7], i.e., true in every world in a Kripke model.

We now describe a systematic procedure for **KEM**. First we define the following notions. Given a branch θ of a **KEM**-tree, we shall call an L-formula $X : u$ *E-analysed in* θ if either (i) X is of type α and both $\alpha_1 : t$ and $\alpha_2 : u$ occur in θ; or (ii) X is of type β and one of the following conditions is satisfied: (a) if $\beta_1^C : v$ occurs in θ and $[u;v]\sigma$, then also $\beta_2 : [u;v]\sigma$ occurs in θ, (b) if $\beta_2^C : v$ occurs in θ and $[u;v]\sigma$, then also $\beta_1 : [u;v]\sigma$ occurs in θ; or (iii) X is of type μ and $\mu_0 : (u', u)$ occurs in θ for some appropriate u' of the right type, not previously occurring in θ, or (iv) X is of type γ and $\gamma_0(x_n) : u$ occurs in θ for some variable x_n not previously occurring in θ or (v) X is of type δ and $\delta_0(c_n) : u$ occurs in θ for some variable c_n not previously occurring in θ.

We shall call a branch θ of a **KEM**-tree *E-completed* if every L-formula in it is E-analysed and it contains no complementary formulas which are not $\sigma_{\mathbf{L}}$-complementary. We shall say a branch θ of a **KEM**-tree *completed* if it is E-completed and all the L-formulas of type β in it either are analysed or cannot be analysed. We shall call a **KEM**-tree *completed* if every branch is completed.

The following procedure starts from the 1-branch, 1-node tree consisting of $\{X_1 : u, \ldots, X_m : v\}$ and applies the inference rules until the resulting **KEM**-tree is either closed or completed.

At each stage of proof search (i) we choose an open non completed branch θ. If θ is not E-completed, then (ii) we apply the 1-premise rules until θ becomes E-completed. If the resulting branch θ' is neither closed nor completed, then (iii) we apply the 2-premise rules until θ becomes E-completed. If the resulting branch θ' is neither closed nor completed, then (iv) we choose an L-formula of type β which is not yet analysed in the branch and apply *PB* so that the resulting *LS*-formulas are $\beta_1 : u'$ and $\beta_1^C : u'$ (or, equivalently $\beta_2 : u'$ and $\beta_2^C : u'$), where $u = u'$ if u is restricted (and already occurring when $h(u) \in \Phi_C$), otherwise u' is obtained from u by instantiating $h(u)$ to a constant not occurring in u; (v) ("Modal *PB*") if the branch is not E-completed nor closed, because of complementary formulas which are not $\sigma_{\mathbf{L}}$-complementary, then we have to see whether a restricted label unifying with both the labels of the complementary formulas occurs previously in the branch; if such a label exists, or can be built using already existing labels and the unification rules, then the branch is closed, (vi) we repeat the procedure in each branch generated by *PB*.

1. $\mathbf{F}\Box_j\Box_p appt$	w_0	9. $\mathbf{T}(place \wedge time \rightarrow appt)$	(W_1^j, W_1^p, w_0)
2. $\mathbf{T}\Box_p\Box_j(place \wedge time \rightarrow appt)$	W_0	10. $\mathbf{F}place \wedge time$	(w_1^p, w_1^j, w_0)
3. $\mathbf{T}\Box_w(\Box_p time \rightarrow \Box_j time)$	W_0	11. $\mathbf{T}\Box_p time \rightarrow \Box_j time$	(W_1^w, w_0)
4. $\mathbf{T}\Box_p\Box_j place$	W_0	12. $\mathbf{T}\Box_j place$	(W_2^p, w_0)
5. $\mathbf{T}\Box_p time$	W_0	13. $\mathbf{T}place$	(W_2^j, W_2^p, w_0)
6. $\mathbf{F}\Box_p appt$	(w_1^j, w_0)	14. $\mathbf{F}time$	(w_1^p, w_1^j, w_0)
7. $\mathbf{F}appt$	(w_1^p, w_1^j, w_0)	15. $\mathbf{T}\Box_p time$	(w_1^j, w_0)
8. $\mathbf{T}\Box_j(place \wedge time \rightarrow appt)$	(W_1^p, w_0)	16. $\mathbf{T}time$	(W_3^p, w_1^j, w_0)
		$\times$	

Fig. 7. Proof of $\Box_j\Box_p$ using **KEM** representation.

Fig.7. shows a **KEM** tableaux proof using the inference rules in section 4.3 and following the proof search mentioned above to solve the first conjunct of (2). The proof goes as follows; 1. is the negation of the formula to be proved. The formulas in 2–5 are the global assumptions of the scenario and accordingly they must hold in every world of every model for it. Hence we label them with a variable W_0 that can unify with every other label. This is used to derive 12. from 11. and 5. using a β-rule, and for introducing 15.; 6. is from 1., and 7. from 6. by applying π rule. Similarly we get 8. from 2., 9. from 8. using ν rule. 10. comes from 9. and 7. through the use of modus tollens. Applying ν rule twice we can derive 11. from 3. as well as 13. from 12. Through propositional reasoning we get 14. from 10. and by a further use of ν rule on 15. we get 16. (14. and 16.) are complementary formulas indicating a contradiction and this results in a closed tableaux because the labels in 14. and 16. unify, denoting that the contradiction holds *in the same world*.

5 Concluding Remarks

In this paper we have argued that BDI logics can be explained in terms of fibring as combination of simpler modal logics. Then we have outlined three labelled tableaux systems (path, graph and unification). For each of the method we have seen how they can deal with the Friend's puzzle as a way to evaluate their features. The path approach requires the definition of new inference rules for each logic, but then we can use a simple labelling mechanism. However, it is not clear how this approach can be extended to more complex cases of fibring, for example when we consider non-normal modal operators for the mental attitudes of the agents.

The graph approach on the other hand does not require, in general, any new rule, since it uses the semantic structure to propagate formulas to the appropriate labels. It is then suitable for an approach based on fibring, since the relationships between two labels can be given in terms of fibring. However, when the structure of the model is more complicated (for example when the models for the logics are given in terms of neighbourhood models) then the approach might not be applicable since it assumes relationships between labels/worlds in a model and not more complex structures. In addition, the system does not give a decision procedure unless the relationships among labels are restricted to decidable fragments of first-order logic. Thus it is not possible to represent logic that are not first-order definable and the designer of an agent logic has to verify that she is operating within a decidable fragment of first order logic.

KEM, in general similar to the graph approach, does not need logic dependent rules, however, similar to the path approach, it needs logic dependant label unifications. We have seen that the label algebra can be seen as a form of fibring [9], thus simple fibring does not require special attention in **KEM**; therefore it allows for a seamless composition of (sub)tableaux for modal logics. The label algebra contrary to the graph reasoning mechanism is not based on first order logic and thus can deal with complex structure and is not limited to particular fragment. Indeed **KEM** has been proved able to deal with complex label schema for non-normal modal logics in a uniform way [11] as well as other intensional logics such as conditional logics [2]. For these reasons we believe that **KEM** offers a suitable framework for decision procedure for multi-modal logic for multi-agent systems. As we only described the static fragment of BDI logics, (no temporal evolution was considered), the future work is to extend the tableaux framework so as to accomodate temporal modalities.

Acknowledgements

This work was supported by the Australian Research Council (ARC) under Discovery Project No. DP0452628 on “Combining Modal Logics for Dynamic and Multi-Agent Systems”.

References

1. A. Artosi, P. Benassi, G. Governatori, and A. Rotolo. Shakespearian modal logic: A labelled treatment of modal identity. In *Advances in Modal Logic*, volume 1. CSLI, 1998.
2. A. Artosi, G. Governatori, and A. Rotolo. Labelled tableaux for non-monotonic reasoning: Cumulative consequence relations. *Journal of Logic and Computation*, 12(6):1027–1060, 2002.
3. Matteo Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension*. PhD thesis, Universita degli Studi di Torino, Italy, 1998.
4. Patrick Blackburn and Maarten de Rijke. Zooming in, zooming out. *Journal of Logic, Language and Information*, 1996.
5. Patrick Blackburn and Martin de Rijke. Why combine logics. *Studia Logica*, 59(1), 1997.
6. Artur S. d’ Avila Garcez and Dov M. Gabbay. Fibring neural networks. In *AAAI-2004*, pages 342–347. AAAI/MIT Press, 2004.
7. Melvin Fitting. *Proof Methods for Modal and Intuitionistic Logics*. Reidel, Dordrecht, 1983.
8. Dov M. Gabbay. *Fibring Logics*. Oxford University Press, Oxford, 1999.
9. Dov M. Gabbay and Guido Governatori. Fibred modal tableaux. In *Labelled Deduction*. Kluwer academic Publishers, 2000.
10. Guido Governatori. Labelled tableau for multi-modal logics. In *TABLEAUX*, volume 918, pages 79–94. Springer, 1995.
11. Guido Governatori and Alessandro Luppi. Labelled tableaux for non-normal modal logics. In *AI*IA 99: Advances in AI*, LNAI-1792, pages 119–130, Berlin, 2000. Springer.
12. Guido Governatori, Vineet Padmanabhan, and Abdul Sattar. On Fibring Semantics for BDI Logics. In *Logics in Artificial Intelligence: (JELIA-02), Italy*, LNAI-2424. Springer, 2002.
13. Marcus Kracht and Frank Wolter. Properties of independently axiomatizable bimodal logics. *The Journal of Symbolic Logic*, 56(4):1469–1485, 1991.
14. John W. Llyod. Modal higher-order logic for agents. http://users.rsise.anu.edu.au/ jwl/beliefs.pdf, 2004.
15. Alessio Lomuscio. *Information Sharing Among Ideal Agents*. PhD thesis, School of Computer Science, University of Brimingham, 1999.
16. Vineet Padmanabhan. *On Extending BDI Logics*. PhD thesis, School of Information Technology, Griffith University, Brisbane, Australia, 2003.
17. Anand S. Rao and Michael P. Georgeff. Formal models and decision procedures for multi-agent systems. Technical note 61, Australian Artificial Intelligence Institute, 1995.
18. Amiílcar Sernadas, Cristina Sernadas, and Carlos Caleriro. Fibring of logics as a categorial construction. *Journal of Logic and Computation*, 9(2):149–179, 1999.
19. Amílcar Sernadas, Cristina Sernadas, and A. Zanardo. Fibring modal first-order logics: Completeness preservation. *Logic Journal of the IGPL*, 10(4):413–451, 2002.
20. Frank Wolter. Fusions of modal logics revisited. In *Advances in Modal Logic*, volume 1. CSLI Lecture notes 87, 1997.
21. Frank Wolter. The decision problem for combined (modal) logics. Technical report, Institut fur Informatik, universitat Leipzig, Germany, www.informatik.uni-leipzig.de/ wolter/, September 9, 1999.
22. A. Zanardo, Amiílcar Sernadas, and Cristina Sernadas. Fibring: Completeness preservation. *Journal of Symbolic Logic*, 66(1):414–439, 2001.

Programming Declarative Goals Using Plan Patterns

Jomi Hübner[1], Rafael H. Bordini[2], and Michael Wooldridge[3]

[1] University of Blumenau (Brazil)
`jomi@inf.furb.br`

[2] University of Durham (UK)
`R.Bordini@durham.ac.uk`

[3] University of Liverpool (UK)
`mjw@csc.liv.ac.uk`

Abstract. AgentSpeak is a well-known language for programming intelligent agents which captures the key features of reactive planning systems in a simple framework with an elegant formal semantics. However, the original language is too abstract to be used as a programming language for developing multi-agent system. In this paper, we address one of the features that are essential for a pragmatical agent programming language. We show how certain *patterns* of AgentSpeak plans can be used to define various types of declarative goals. In order to do so, we first define informally how plan failure is handled in the extended version of AgentSpeak available in ***Jason***, a Java-based interpreter; we also define special (internal) actions used for dropping intentions. We present a number of *plan patterns* which correspond to elaborate forms of declarative goals. Finally, we give examples of the use of such types of declarative goals and describe how they are implemented in ***Jason***.

1 Introduction

The AgentSpeak(L) language, introduced by Rao in 1996, provides a simple and elegant framework for intelligent action via the run-time interleaved selection and execution of plans. Since the original language was proposed, substantial progress has been made both on the theoretical foundations of the language (e.g., its formal semantics [6]), and on its use, via implementations of practical extensions of AgentSpeak [5]. However, one problem with the original AgentSpeak(L) language is that it lacks many of the features that might be expected by programmers in practical development. Our aim in this paper is to focus on the integration of one such features, namely the definition of declarative goals and the use of plan patters. Throughout the paper, we use AgentSpeak as a more general reference to AgentSpeak(L) and its extensions.

In this paper, we consider the use of *declarative goals* in AgentSpeak programming. By a declarative goal, we mean a goal that *explicitly* represents a state of affairs to be achieved, in the sense that, if an agent has a goal $p(t_1, \ldots, t_n)$, it expects to eventually believe $p(t_1, \ldots, t_n)$ (cf. [19]) and only then can the goal be considered achieved. Moreover, we are interested not only in goals representing states of affairs, but goals that may have complex temporal structures. Currently, although goals form a central component of AgentSpeak programming, they are only *implicit* in the plans defined by the agent programmer. For example, there is no explicit way of expressing that a goal should be maintained until a certain condition holds; such temporal goal structures are defined implicitly, within the plans themselves, and by *ad hoc* efforts on the part of programmers.

While one possibility would be to extend the language and its formal semantics to introduce an explicit notion of declarative goal (as done in other languages, e.g., [19, 7, 22]), we show that this is unnecessary. We introduce a number of *plan patterns*, corresponding to common types of explicit temporal (declarative) goal structures, and show how these can be mapped into AgentSpeak code. Thus, a programmer or designer can conceive of a goal at the declarative level, and this goal will be expanded, via these patterns, into standard AgentSpeak code. We then show how such goal patterns can be used in ***Jason***, a Java-based implementation of an extended version of AgentSpeak [4].

In order to present the plan patterns that can be used for defining certain types of declarative goals discussed in the literature, the *plan failure* handling mechanism implemented in ***Jason***, and some pre-defined *internal actions* used for dropping goals, need to be presented. Being able to handle plan failure is useful not only in the context of defining plan patterns that can represent complex declarative goals. In most practical scenarios, plan failure is not only possible, it is commonplace: a key component of rational action in humans is the ability to handle such failures. After presenting these features of ***Jason*** that are important in controlling the execution of plans, we can then show the plan patterns that define more complex types of goals than has been claimed to be possible in AgentSpeak [7]. We present (declarative) maintenance as well as achievement goals, and we present different forms of commitments towards goal achievement/maintenance (e.g., the well-known blind, single-minded, and open-minded forms of commitment [18]). Finally, we discuss ***Jason*** implementations of examples that appeared in the literature on declarative goals; the examples also help in showing why declarative goals with complex temporal structures are an essential feature in programming multi-agent systems.

2 Goals and Plans in AgentSpeak

In [17], Rao introduced the AgentSpeak(L) programming language. It is a natural extension of logic programming for the BDI agent architecture, and provides an elegant abstract framework for programming BDI agents. In this paper, we only give a very brief introduction to AgentSpeak; see e.g. [6] for more details.

An AgentSpeak agent is created by the specification of a set of initial beliefs and a set of plans. A *belief atom* is simply a first-order predicate in the usual notation, and belief atoms or their negations are *belief literals*. The initial beliefs define the state of the belief base at the moment the agent starts running; the belief base is simply a collection of ground belief atoms (or, in ***Jason***, literals).

AgentSpeak distinguishes two types of goals: *achievement goals* and *test goals*. Achievement goals are predicates (as for beliefs) prefixed with the '`!`' operator, while test goals are prefixed with the '`?`' operator. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these lead to the execution of other plans.) A *test goal* states that the agent wants to test whether the associated predicate is a belief (i.e., whether it can be unified with one of the agent's beliefs).

Next, the notion of a *triggering event* is introduced. It is a very important concept in this language, as triggering events define which events may initiate the execution of plans; the idea of *event*, both internal and external, will be made clear below. There are two types of triggering events: those related to the *addition* ('+') and *deletion* ('-') of mental attitudes (beliefs or goals).

Plans refer to the *basic actions* that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called *action symbols*) used to distinguish them. The actual syntax of AgentSpeak programs is based on the definition of plans, as follows. If e is a triggering event, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then e **:** b_1 **&** ... **&** $b_m \leftarrow h_1$ **;** ... **;** h_n**.** is a *plan*.

An AgentSpeak(L) plan has a *head* (the expression to the left of the arrow), which is formed from a triggering event (denoting the purpose for that plan), and a conjunction of belief literals representing a *context* (separated from the triggering event by ':'). The conjunction of literals in the context must be satisfied if the plan is to be executed (the context must be a logical consequence of that agent's current beliefs). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered.

Besides the belief base and the plan library, the AgentSpeak interpreter also manages a set of *events* and a set of *intentions*, and its functioning requires three *selection functions*. The event selection function selects a single event from the set of events; another selection function selects an "option" (i.e., an applicable plan) from a set of applicable plans; and a third selection function selects one particular intention from the set of intentions. The selection functions are supposed to be agent-specific, in the sense that they should make selections based on an agent's characteristics in an application-specific way. An event has the form $\langle te, i \rangle$, where te is a plan triggering event (as in the plan syntax described above) and i is that intention that generated the event or T for external events.

Intentions are particular courses of actions to which an agent has committed in order to handle certain events. Each intention is a stack of partially instantiated plans. Events, which may start the execution of plans that have relevant triggering events, can be *external*, when originating from perception of the agent's environment (i.e., addition and deletion of beliefs based on perception are external events); or *internal*, when generated from the agent's own execution of a plan (i.e., a subgoal in a plan generates an event of type "addition of achievement goal"). In the latter case, the event is accompanied with the intention which generated it (as the plan chosen for that event will be pushed on top of that intention). External events create new intentions, representing separate focuses of attention for the agent's acting within the environment.

3 Plan Failure

We identify three cases of plan failure. The first cause of failure is a *lack of relevant or applicable plans*, which can be understood as the agent "not knowing how to do something". This happens either because the agent simply does not have the know-how (in case it has no relevant plans) — this could happen through simple omission (the programmer did not provide any appropriate plans) — or because all known ways of achieving the goal cannot currently be used (there are known plans but whose contexts do not match the agent's current beliefs). The second is where a test goal fails; that is, where the agent "expected" to believe in a certain condition of the world, but in fact the condition did not hold. The third is where an internal action ("native method"), or a basic action (the effectors within the agent architecture are assumed to provide feedback to the interpreter stating whether the requested action was executed or not), fails.

Regardless of the reason for a plan failing, the interpreter generates a goal deletion event (i.e., an event for "$-!g$") if the corresponding goal achievement ($+!g$) has failed. This paper introduces for the first time an (informal) semantics for the notion of goal deletion as used in ***Jason***. In the original definition, Rao syntactically defined the possibility of goal deletions as triggering events for plans (i.e., triggering event with **`-!`** and **`-?`** prefixes), but did not discuss what they meant. Neither was goal deletion discussed in further attempts to formalise AgentSpeak or its ancestor dMars [12, 11]. Our own choice was to use this as some kind of plan failure handling mechanism[4], as discussed below (even though this was probably not what they originally were intended for).

The idea is that a plan for a goal deletion is a "clean-up" plan, executed prior to (possibly) "backtracking" (i.e., attempting another plan to achieve the goal for which a plan failed). One of the things programmers might want to do within the goal deletion plan is to attempt again to achieve the goal for which the plan failed. In contrast to conventional logic programming languages, during the course of executing plans for subgoals, AgentSpeak programs generate a sequence of actions that the agent performs on the external environment so as to change it, the effects of which cannot be undone by simply backtracking (i.e., it may require further action in order to do so). Therefore, in certain circumstances one would expect the agent to have to "undo" the effects of certain actions before attempting some alternative courses of action to achieve that goal, and this is precisely the practical use of plans with goal deletions as triggering events.

It is important to observe that omitting possible goal deletion plans for existing goal additions implicitly denotes that such goal should never be backtracked, i.e., no alternative plan for it should be attempted in case one fails. To specify that backtracking should always be attempted (e.g., until special internal actions in the plan explicitly cause the intention to be dropped), all the programmer has to do is to specify a goal deletion plan (for a given goal g addition) with empty context and the same goal in the body, as in "`-!`g`: true` $\leftarrow$ `!`g`.`".

[4] The notation $-!g$, i.e., "goal deletion" also makes sense for such plan failure mechanism; if a plan fails there is a possibility that the agent may need to drop the goal altogether, so it is to handle such event (of the possible need to drop a goal) that plans of the form $-!g : \ldots$ are written.

When a failure happens, the whole intention is dropped if the triggering event of the plan being executed was neither an achievement nor a test goal *addition*: only these can be attempted to recover from failure using the goal deletion construct (one cannot have a goal deletion event posted for a failure in a goal deletion plan). In any other circumstance, a failed plan means that the whole intention cannot be achieved. If a plan for a goal addition (`+!`g) fails, the intention i where that plan appears is suspended, and the respective goal deletion event ($\langle -!g, i\rangle$) is included in the set of events. Eventually, this might lead to the goal addition being attempted again as part of the plan to handle the `-!`g event. When the plan for `-!`g finishes not only itself but also the failed `+!`g plan below it[5] are removed from the intention. As it will be clear later, it is a programmer's decision to attempt the goal again or not, or even to drop the whole intention (possibly with special internal action constructs, whose informal semantics is given below), depending on the circumstances. What happens when a plan fails is shown in Figure 1.

```
+!g1(t):  ct
  <- a(t);
     !g2(t);
     ?g2(t);
     ... .
-----------------
te  : ct
  <- !g1(t);
     ... .
```

(a) An Intention before Plan Failure

```
-!g1(t):  ct
  <- ... ;
     !g1(t);
     ... .
-----------------
+!g1(t):  ct
  <- a(t);
     !g2(t);
     ?g2(t);
     ... .
-----------------
te  : ct
  <- !g1(t);
     ... .
```

(b) That Intention after Plan Failure

Fig. 1. Plan Failure.

In the circumstance described in Figure 1(a) above, suppose $a(t)$ fails, or otherwise after that action succeeds an event for $+!g2(t)$ was created but there were no applicable plans to handle the event, or $?g2(t)$ is not is the belief base, nor there are applicable plans to handle a $+?g2(t)$ event. In any of those cases, the intention is suspended and an event for $-!g1(t)$ is generated. Assuming the programmer included a plan for $-!g1(t)$, and the plan is applicable at the time the event is selected, the intention will eventually look as in Figure 1(b). Otherwise the original goal addition event is re-posted or the whole intention dropped, depending on a setting of the ***Jason*** interpreter that is configurable by programmers. (See [1] for an overview of how various BDI systems deal with the problem of there being no applicable plans.)

The reason why not providing goal deletion plans in case a goal is not to be backtracked works is because an event (with the whole suspended intention within it) is discarded in case there are no relevant plans for a generated goal deletion. In general, the lack of relevant plans for an event indicates that the perceived event is not significant for the agent in question, so they are simply ignored. An alternative approach for handling the lack of relevant plans is described in [2], where it is assumed that in some cases, explicitly specified by the programmer, the agent will want to ask other agents how to

5 The failed plan is left in the intention, for example, so that programmers could check which plan failed (e.g., by means of ***Jason*** internal actions).

handle such events. The mechanism for plan exchange between AgentSpeak agents presented in [2] allows the programmer to specify which triggering events should generate attempts to retrieve external plans, which plans an agent agrees to share with others, what to do once the plan has been used for handling that particular event instance, and so on.

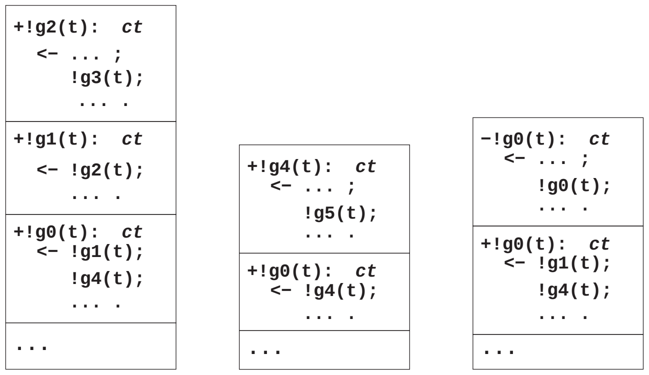
In the next section, besides the plan failure handling mechanism, we also make use of a particular standard internal action. Standard internal actions, unlike user-defined internal actions, are those available with the ***Jason*** distribution; they are denoted by an action name starting with symbol '`.`'. Some of these pre-defined internal actions manipulate the structure used in giving semantics to the AgentSpeak interpreter. For that reason, they need to be precisely defined. As the focus here is on the use of patterns for defining declarative goals, we will give only informal semantics to the internal action we refer to in the next section.

The particular internal action used in this paper is `.dropGoal(`g`,true)`. Any intention that has the goal g in the triggering event of any of its plans will be changed as follows. The plan with triggering event `+!`g is removed and the plan below that in the stack of plans forming that intention carries on being executed at the point after goal g appeared. Goal g, as it appears in the `.dropGoal` internal action is used to further instantiate the plan where the goal that was terminated early appears. With `.dropGoal(`g`,false)`, the plan for `+!`g is also removed, but an event for the deletion of the goal whose plan body required g is generated: this informally means that there is no way of achieving g so the plan requiring g to be achieved must fail. That is, `.dropGoal(`g`,true)` is used when the agent realises the goal has already been achieved so whatever plan was being executed to achieve that goal does not need to be executed any longer. On the other hand, `.dropGoal(`g`,false)` is used when the agent realises that the goal has become impossible to achieve, hence the need to fail the plan that required g being achieved as one of its subgoals.

It is perhaps easier to understand how these actions work with reference to Figure 2. The figure shows the consequence of each of these internal actions being executed (the plan where the internal action appeared is not shown; it is likely to be within another intention). Note that the state of the intention affected by the execution of one of these internal actions, as shown in the figure, is not the immediate resulting state (at the end of the reasoning cycle where the internal action was executed) but the most significant next state of the changed intention.

4 Declarative Goal Patterns

Although goals form a central component of the AgentSpeak conceptual framework, it is important to note that the language itself does not provide any explicit constructs for handling goals with complex temporal structure. For example, a system designer and programmer will often think in terms of goals such as "maintain P until Q becomes true", or "prevent P from becoming true". Creating AgentSpeak code to realise such complex goals has, to date, been largely an *ad hoc* process, dependent upon the experience of the programmer. Our aim in this section is firstly to define a number of declarative goal structures, and secondly to show how these can be realised in terms



```
+!g2(t):  ct
  <- ... ;
     !g3(t);
      ... .
+!g1(t):  ct
  <- !g2(t);
     ... .
+!g0(t):  ct
  <- !g1(t);
     !g4(t);
     ... .
...
```

```
+!g4(t):  ct
  <- ... ;
     !g5(t);
     ... .
+!g0(t):  ct
  <- !g4(t);
     ... .
...
```

```
-!g0(t):  ct
  <- ... ;
     !g0(t);
     ... .
+!g0(t):  ct
  <- !g1(t);
     !g4(t);
     ... .
...
```

(a) Initial Intention (b) After `.dropGoal(g1(t),true)` (c) After `.dropGoal(g1(t),false)`

Fig. 2. Standard Internal Actions for Dropping Goals.

of *patterns* of AgentSpeak plans — that is, complex combinations of plan structures which are often useful in actual scenarios. As we shall see, such patterns can be used to implement, in a systematic way, not only complex types of declarative goals, but also the types of commitments they represent, as discussed for example by Cohen and Levesque [8].

As an initial motivational example for declarative goals, consider a robot agent with the goal of being at some location (represented by the predicate `l(`X`,`Y`)`) and the following plan to achieve this goal:

$$\texttt{+!l(X,Y): bc(B) \& B} > 0.2 \leftarrow \texttt{go(X,Y).}$$

where the predicate `bc/1` stands for "battery charge", and `go` identifies an action that the robot is able to perform in the environment.

At times, using an AgentSpeak plan as a procedure, can be a quite useful programming tool. Thus, in a way, it is important that the AgentSpeak interpreter does not enforce any declarative semantics to its only (syntactically defined) goal construct. However, in the plan above, `l(`X`,`Y`)` is clearly meant as a declarative goal; that is, the programmer expects the robot to believe `l(`X`,`Y`)` (by perceiving the environment) if the plan executes to completion. If it fails because, say, the environment is dynamic, the goal cannot be considered achieved and, normally, should be attempted again.

This type of situation is commonplace in multi-agent system, and this is why it is important to be able to define declarative goals in agent-oriented programming. However, this can be done without the need to change the language and/or its semantics. As similarly pointed out by van Riemsdijk *et al.* [19], we can easily transform the above procedural goal into a declarative goal by adding a corresponding *test goal* at the end of the plan's body, as follows:

```
+!l(X,Y): bc(B) & B > 0.2 ← go(X,Y); ?l(X,Y).
```

This plan only succeeds if the goal is actually (believed to be) achieved; if the given (procedural) plan executes to completion (i.e., without failing) but the goal happens not to be achieved, the test goal at the end will fail. In this way, we have taken a simple *procedural* goal and transformed it into a *declarative* goal – the goal to achieve some state of affairs.

This solution forms a plan pattern, which can be applied to solve other similar problems which, as we mention above, are commonplace in agent programming. Thus, our approach to include declarative goals in AgentSpeak programming is inspired by the successful adoption of design patterns in object oriented design [13]. To represent such patterns for AgentSpeak, we shall make use of skeleton programs with meta variables. For example, the general form of an AgentSpeak plan for a simple declarative goal, as the one used in the robot's location goal above, is as follows:

+!g: c $\leftarrow$ p; ?g.

Here, g is a meta variable that represents the declarative goal, c is a meta variable that represents the context expression stating in which circumstances the plan is applicable, and p represents the procedural part of the plan body (i.e., a course of action to achieve g). Note that, with the introduction of the final test goal, this plan to achieve g finishes successfully only if the agent believes g after the execution of plan body p.

To simplify the use of the patterns, we also define pattern rules which rewrite a set of AgentSpeak plans into a new AgentSpeak program according to a given pattern.[6] The following pattern rule, called **DG** (Declarative Goal), is used to transform procedural goals into declarative goals. The pattern rule name is followed by the parameters which need to be provided by the programmer, besides the actual code (i.e., a set of plans) on which the pattern will be applied.

$$\frac{\begin{array}{l}\texttt{+!}g\texttt{: } c_1 \leftarrow p_1\texttt{.}\\ \texttt{+!}g\texttt{: } c_2 \leftarrow p_2\texttt{.}\\ \ldots\\ \texttt{+!}g\texttt{: } c_n \leftarrow p_n\texttt{.}\end{array}}{\begin{array}{l}\texttt{+!}g\texttt{: } g \leftarrow \texttt{true.}\\ \texttt{+!}g\texttt{: } c_1 \leftarrow p_1\texttt{; ?}g\texttt{.}\\ \texttt{+!}g\texttt{: } c_2 \leftarrow p_2\texttt{; ?}g\texttt{.}\\ \ldots\\ \texttt{+!}g\texttt{: } c_n \leftarrow p_n\texttt{; ?}g\texttt{.}\\ \texttt{+}g\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, true).}\end{array}}\ \mathbf{DG}_g\ (n \geq 1)$$

Essentially, this rule adds ?g at the end of each plan in the given set of plans which has +!g as trigger event, and creates two extra plans (the first and the last plans above). The

[6] Note that some of the patterns presented in this paper require the atomic execution of certain plans, but we avoid including this in the patterns for clarity of presentation; this feature is available in ***Jason*** through a simple plan annotation.

first plan checks whether the goal g has already been achieved — in such case, there is nothing else to do. That last plan is triggered when the agent perceives that g has been achieved while it is executing any of the courses of action p_i ($1 \leq i \leq n$) which aim at achieving g; in this circumstance, the plan being executed in order to achieve g can be immediately terminated. The internal action `.dropGoal(`g`, true)` terminates such plan with success (as explained in Section 3).

In this pattern, when one of the plans to achieve g fails, the agent gives up achieving the goal altogether. However it could be the case that for such goal, the agent should try another plan to achieve it, as in the "backtracking" plan selection mechanism available in platforms such as JACK [21, 14] and 3APL [10, 9]. In those mechanisms, usually only when all available plans have been tried in turn and failed is the goal abandoned with failure, or left to be attempted again later on. The following rule, called **BDG** (Backtracking Declarative Goal), defines this pattern based on a set of conventional AgentSpeak plans $\mathcal{P}$ transformed by the **DG** pattern (each plan in $\mathcal{P}$ is of the form `+!`g`:` $c \leftarrow p$):

$$\frac{\mathcal{P}}{\begin{array}{l}\mathbf{DG}_g(\mathcal{P})\\ \texttt{-!}g\texttt{: true} \leftarrow \texttt{!}g\texttt{.}\end{array}}\ \mathbf{BDG}_g$$

The last plan of the pattern catches a failure event, caused when a plan from $\mathcal{P}$ fails, and then tries to achieve that same goal g again. Notice that it is possible that the same plan is selected and fails again, causing a loop if the plan contexts have not been carefully programmed. Thus the programmer would need to specify the plan contexts in such a way that a plan is only applicable if it has a chance of succeeding regardless of it having been tried already (recently).

Instead of worrying about defining contexts in such more general way, in some cases it may be useful for the programmer to apply the following pattern, called **EBDG** (Exclusive BDG), which ensures that none of the given plans will be attempted twice before the goal is achieved:

$$\frac{\begin{array}{l}\texttt{+!}g\texttt{:}\ c_1 \leftarrow b_1\texttt{.}\\ \texttt{+!}g\texttt{:}\ c_2 \leftarrow b_2\texttt{.}\\ \ldots\\ \texttt{+!}g\texttt{:}\ c_n \leftarrow b_n\texttt{.}\end{array}}{\begin{array}{l}\texttt{+!}g\texttt{:}\ g \leftarrow \texttt{true.}\\ \texttt{+!}g\texttt{: not p1(}g\texttt{) \& }c_1 \leftarrow \texttt{+p1(}g\texttt{); }b_1\texttt{.}\\ \texttt{+!}g\texttt{: not p2(}g\texttt{) \& }c_2 \leftarrow \texttt{+p2(}g\texttt{); }b_2\texttt{.}\\ \ldots\\ \texttt{+!}g\texttt{: not p}n\texttt{(}g\texttt{) \& }c_n \leftarrow \texttt{+p}n\texttt{(}g\texttt{); }b_n\texttt{.}\\ \texttt{-!}g\texttt{: true} \leftarrow \texttt{!}g\texttt{.}\\ \texttt{+}g\texttt{: true} \leftarrow \texttt{-p1(}g\texttt{); -p2(}g\texttt{); }\ldots\texttt{ .dropGoal(}g\texttt{, true).}\end{array}}\ \mathbf{EBDG}_g$$

In this pattern, each plan, when selected for execution, initially adds a belief `p`i`(`g`)`; the goal g is used as an argument to `p` so as to avoid interference among applications of the pattern for different goals. The belief is used as part of the plan contexts (note the use of `not p`i in the contexts of the plans in the pattern above) to state the plan should not be applicable in a second attempt (of that same plan within a single adoption of goal g for that agent).

In the pattern above, despite the various alternative plans, the agent can still end up dropping the intention with the goal g unachieved, if all those plans become non-applicable. Conversely, in a *blind commitment goal* the agent can drop the goal only when it is achieved. This type of commitment toward the achievement of a declarative goal can thus be understood as *fanatical commitment* [18]. The **BCG**$_{g,F}$ pattern below defines this type of commitment:

$$\frac{\mathcal{P}}{\begin{array}{l}\boldsymbol{F}(\mathcal{P}) \\ \texttt{+!}g\texttt{: true} \leftarrow \texttt{!}g\texttt{.}\end{array}} \ \mathbf{BCG}_{g,F}$$

This pattern is based on another pattern rule (represented by the variable ***F***); ***F*** is often **BDG**, although the programmer can chose another pattern (e.g., **EBDG** if a plan should not be attempted twice). Finally, the last plan keeps the agent pursuing the goal even in case there is no applicable plan. It is assumed that the selection of plans is based on the order that the plans appear in the program and all events have equal chance of being chosen as the event to be handled in a reasoning cycle.

For most applications, **BCG**-style fanatical commitment is too strong. For example, if a robot has the goal to be at some location, it is reasonable that it can drop this goal in case its battery charge is getting very low; in other words, the agent has realised that it has become impossible to achieve the goal, so it is useless to keep attempting it. This is very similar to the idea of a persistent goal in the work of Cohen and Levesque: a persistent goal is a goal that is maintained as long as it is believed not achieved, but still believed possible [8]. In [22] and [7], the “impossibility” condition is called a “drop condition”. The drop condition f (e.g., “low battery charge”) is used in the Single-Minded Commitment (**SMC**) pattern to allow the agent to drop a goal if it becomes impossible:

$$\frac{\mathcal{P}}{\begin{array}{l}\mathbf{BCG}_{g,BDG}(\mathcal{P}) \\ \texttt{+}f\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, false).}\end{array}} \ \mathbf{SMC}_{g,f}$$

This pattern extends the **BCG** pattern adding the drop condition represented by the literal f in the last plan. If the agent comes to believe f, it can drop goal g, signalling failure (refer to the semantics of the internal action `.dropGoal` in section 3). This effectively means that the plan in the intention where g appeared, which depended on g to carry on execution, must itself fail (as g is now impossible to achieve). However, there might be an alternative for that other plan which does not depend on g, so that plan’s failure handling may take care of such situation.

As we have a failure drop condition for a goal, we can also have a successful drop condition, e.g., because the motivation to achieve the goal has ceased to exist. Suppose a robot has the goal of going to the fridge because its owner has asked it to fetch a beer from there; then, if the robot realises that its owner does not want a beer anymore, it should drop the goal [8]. The belief "my owner wants a beer" is the *motivation* m for the goal. The following pattern, called Relativised Commitment Goal (**RCG**) defines a goal that is relative to a motivation condition: the goal can be dropped with success if the agent looses the motivation for it.

$$\frac{\mathcal{P}}{\begin{array}{l}\mathbf{BCG}_{g,BDG}(\mathcal{P})\\ \texttt{-}m\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, true).}\end{array}}\ \mathbf{RCG}_{g,m}$$

Note that, in the particular combination of **RCG** and **BCG** above, if the attempt to achieve g ever terminates, it will always terminate with success, since the goal will be dropped only if either the agent believes it has been achieved achieved (by **BCG**) or m is removed from belief base.

Of course we can combine the last two patterns above to create a goal which can be dropped if it has been achieved, has become impossible to achieve, or the motivation to achieve it no longer exists (representing an open-minded commitment). The Open-Minded Commitment pattern (**OMC**) defines this type of goal:

$$\frac{\mathcal{P}}{\begin{array}{l}\mathbf{BCG}_{g,BDG}(\mathcal{P})\\ \texttt{+}f\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, false).}\\ \texttt{-}m\texttt{: true} \leftarrow \texttt{.dropGoal(}g\texttt{, true).}\end{array}}\ \mathbf{OMC}_{g,f,m}$$

For example, a drop condition could be "no beer at location (X,Y)" (denoted below by ¬ `b(X,Y)`), and the motivation condition could be "my owner wants a beer" (denoted below by `wb`). Consider the initial plan below with representing the single known course of action to achieve goal `l(X,Y)`:

```
+!l(X,Y): bc(B) & B > 0.2 ← go(X,Y).
```

When the pattern $\mathbf{OMC}_{l(X,Y),\neg b(X,Y),wb}$ is applied to the plan above, we get the following program:

```
+!l(X,Y): l(X,Y) ← true.
+!l(X,Y): bc(B) & B > 0.2 ← go(X,Y); ?l(X,Y).
+!l(X,Y): true ← !l(X,Y).
-!l(X,Y): true ← !l(X,Y).
+¬b(X,Y): true ← .dropGoal(l(X,Y), false).
-wb: true ← .dropGoal(l(X,Y), true).
```

Another important type of goal in agent-based systems are *maintenance goals*: the agent needs to ensure that the state of the world will always be such that g holds. Whenever the agent realises that g is no longer in its belief base (i.e., believed to be true), it attempts to bring about g again by having the respective declarative (achievement) goal. The pattern rule that defines a Maintenance Goal (**MG**) is as follows:

$$\frac{\mathcal{P}}{\begin{array}{l}g\texttt{.}\\ \texttt{-}g\texttt{: true} \leftarrow \texttt{!}g\texttt{.}\\ \boldsymbol{F}(\mathcal{P})\end{array}}\ \mathbf{MG}_{g,F}$$

The first line of the pattern states that, initially (when the agent starts running) it will assume that g is true. (As soon as the interpreter obtains perception of the environment for the first time, the agent might already realise that such assumption was wrong.) The first plan is triggered when g is removed from the belief base, e.g. because g has not been perceived in the environment in a given reasoning cycle, and thus the maintenance goal g is no longer achieved. This plan then creates a declarative goal to achieve g. The type of commitment to achieving g if it happens not to be true is defined by ***F***, which would normally be **BCG** given that the goal should not be dropped in any circumstances unless it is has been achieved again. (Realistically, plans for the agent to attempt pro-actively to prevent this from even happening would also be required, but the pattern is useful to make sure the agent will act appropriately in case things go wrong.)

Another useful pattern is a Sequenced Goal Adoption (**SGA**). This pattern should be used when various instances of a goal should not be adopted concurrently (e.g., a robot that needs to clean two different places). To solve this problem, the **SGA** pattern adopts the first occurrence of the goal and records the remaining occurrences as pending goals by adding them as special beliefs. As one such goal occurrence is achieved, if any other occurrence is pending, it gets activated.

$$\frac{}{\begin{array}{l}t\texttt{: not fl(\_) \& } c \leftarrow \texttt{!fg(}g\texttt{).}\\ t\texttt{: fl(\_) \& } c \leftarrow \texttt{+fl(}g\texttt{).}\\ \texttt{+!fg(}g\texttt{): true} \leftarrow \texttt{+fl(}g\texttt{); !}g\texttt{; -fl(}g\texttt{).}\\ \texttt{-!fg(}g\texttt{): true} \leftarrow \texttt{-fl(}g\texttt{).}\\ \texttt{-fl(\_): fl(}g\texttt{)} \leftarrow \texttt{!fg(}g\texttt{).}\end{array}}\ \mathbf{SGA}_{t,c,g}$$

In this pattern, t is the trigger leading to the adoption of a goal g; c is the context for the goal adoption; `fl(`g`)` is the flag to control whether the goal g is already active; and `fg(`g`)` is a procedural goal that guarantees that `fl` will be added to the belief base to record the fact that some occurrence of the goal has already been adopted, then adopts the goal `!`g, as well as it guarantees that `fl` will be eventually removed whether `!`g succeeds or not. The first plan is selected when g is not being pursued; it simply calls the `fg` goal. The second plan is used if some other instance of that goal has already been adopted. All it does is to remember that this goal g was not immediately adopted by adding `fl(`g`)` to the belief base. The last plan makes sure that whenever a goal adoption instance is finished (denoted by the removal of a `fl` belief), if there are any pending goal instances to be adopted, they will be activated through the `fg` call.

5 Using Patterns in *Jason*

Jason is an interpreter for an extended version of AgentSpeak(L) and is available *Open Source* under GNU LGPL at `http://jason.sourceforge.net` [4]. It implements the operational semantics of AgentSpeak(L) as given in [6]. It also implements the plan failure mechanism and the pre-defined internal action[7] used in the patterns described in Section 4. Since these features are enough for programming declarative goals, ***Jason*** already supports them. However, it would be clearly not acceptable if the programmer had to apply the patterns by hand.

To simplify the programming of sophisticated goals by the use of patterns, we extend the language interpreted by ***Jason*** to include pre-processing directives. The syntax for pattern directives is:

```
directive ::=
  "{" "begin" <pattern-name>"("<parameters>")" "}"
      <agent-speak-program>
  "{" "end" "}"
```

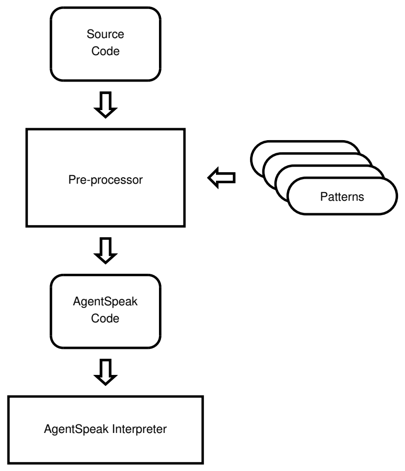


Fig. 3. ***Jason*** Pre-Processing and Patterns.

We have implemented a pre-processor for ***Jason*** which also handles patterns as illustrated in Figure 3. Each pattern is implemented in a Java class that receives an AgentSpeak program and returns another program, transformed as defined by the respective pattern. This implementation allows us, and even users, to make new patterns available in a straightforward manner. One simply has to create a new Java class for the new pattern and register this class with the pre-processor[8].

In the remainder of this section, we will illustrate how the ***Jason*** pre-processing directives for the use of patterns can be used to program a cleaning robot for the scenario described in [7] (where the robot was implemented using Jadex [15, 16]). The first goal of the robot is to maintain its battery charged: this is clearly a maintenance goal (**MG**). The agent should pursue this goal when its battery level goes below 20% and should remain pursuing it until the battery is completely charged. In the program below, based on the perception of the battery level, the belief `battery_charged`, which indicates that the goal is satisfied, is either removed or added to the belief base, signalling whether the corresponding achievement goal must be activated or not.

[7] The internal action used here is not yet available in the latest public release of ***Jason***, but will be available in the next release.

[8] Note that this too will only be available in the next release of ***Jason***

```
+battery_level(B): B < 0.2 ← -battery_charged.
+battery_level(B): B = 1.0 ← +battery_charged.

{ begin mg("battery_charged", bcg("battery_charged")) }
   +!battery_charged : not l(power_supply)
          go(power_supply).
   +!battery_charged: l(power_supply) ← plug_in.
{ end }
```

The first plan of the pattern for the `battery_charged` goal moves the agent to the place where there is a power supply, if it is not already there (according to its `l(power_supply)` belief). Otherwise, the second plan will plug the robot to the power supply. The `plug_in` action will charge the battery and thus change the robot's state that is perceived back through `battery_level(B)` percepts (which generate `+battery_level(B)` events).

The second goal the robot might adopt is to patrol the museum at night. This goal is therefore activated when the agent perceives sunset (represented by the event `+night`). Whenever activated, the goal can be dropped only if the agent perceives dawn (represented by the event `-night`). The following program defines `patrol` as this kind of goal using a **RCG** pattern with `night` as the motivation:

```
+night: true ← !patrol.

{ begin rcg("patrol", "night") }
   +!patrol: battery_charged ← wander.
{ end }
```

The agent will never have the belief `patrol` in its belief base, since no plan or perception of the environment will add this particular belief. The goal is, in some sense, deliberately unachievable, while RCG maintains the agent committed to the goal nevertheless. However, it is considered as achieved (finished with success) when the motivation condition is removed from the belief base. Note that the context for the `!patrol` plan is that the battery is charged, therefore while the maintenance goal `battery_charged` is active, the robot does not wander, but it resumes wandering as soon the battery becomes charged again. We are thus using this belief to create an *interference* between goals (i.e., charging the battery precludes patrolling).

The last goal the robot might adopt is to clean the museum during the day whenever it perceives waste around. Since the robot can perceive various different pieces of waste around, it would accordingly generate several concurrent instances of this goal. However these goals are mutually exclusive: they cannot be achieved simultaneously; trying to go in two different directions must be avoided, and expressing this at the declarative level avoids too much work on implementing application-specific intention selection functions (in the context of AgentSpeak). It is indeed another kind of interference between different goals. The **SGA** pattern is used in the program below to ensure that only one `clean` goal instance is being pursued at a moment in time. The event that triggers this goal is `+waste(X,Y)` (some waste being perceived at location X,Y), and the context is `not night`:

```
{ begin sga("+waste(X,Y)", "not night", "clean(X,Y)")}
{ end }

{ begin omc("clean(X,Y)", "night", "waste(X,Y)")}
   +!clean(X,Y): l(X,Y) ← pick; go(bin); drop.
   +!clean(X,Y): not l(X,Y) ← go(X,Y).
{ end }
+battery_charged: true ← .suspend(clean(X,Y)).
-battery_charged: true ← .resume(clean(X,Y)).
```

In the program above, an open-minded commitment pattern (**OMC**) is used to create the `clean(X,Y)` goal with `night` as the failure condition (at sunset, the goal should be abandoned with failure) and `waste(X,Y)` as the motivation (if the agent came to believe that there is no longer waste at that location, the goal could be dropped with success). The last two plans are used to suspend and resume the goal when the `battery_charge` goal is active. Of course we could add `battery_charge` in the context of the plans (as we did in the `patrol` goal); however, using the `.suspend` internal action is more efficient because the goal becomes actually suspended (until resumed with the respective `.resume` internal action) rather than being continuously attempted without any applicable plans.

6 Conclusions

In this paper we have shown that sophisticated types of goals discussed in the agents literature can be implemented in the AgentSpeak language with only the extensions (and extensibility mechanisms) available in ***Jason***. In fact, this is done by combining AgentSpeak plans, forming certain patterns, for each type of goal and commitment towards goals that agents may have. Therefore, our approach is to take advantage of the simplicity of the AgentSpeak language, using only its well-known support for procedural goals plus the idea of "plan patterns" to support the use of declarative goals with complex temporal structures in AgentSpeak programming.

Besides the use of internal actions such as `.dropGoal` (that are available in ***Jason*** for general use, independently of this proposal for declarative goals), our proposal does not require either: (i) syntactical or semantical changes in the language (as done, for example, in [22, 7]); nor (ii) the definition of a goal base (cf. [19]) which is also usual in other approaches. Van Riemsdijk *et at.* [20] also pointed out that declarative goals can be built based on the procedural goals available in 3APL, by simply checking if the corresponding belief is true at the end of the plan execution. What they proposed in that paper corresponds to our **BDG** pattern. In this work, we further define various other types of declarative goals, represented them as *patterns* of AgentSpeak programs, and presented an implementation in ***Jason*** (using a pre-processor) that facilitates this approach for declarative goals. Another advantage of our approach is that, as complex types of goals are mapped to plain AgentSpeak using patterns, programmers can change the patterns to fit their own requirements, or indeed create new patterns easily.

In future work we intend to formalise our approach based on the existing operational semantics and to verify some properties of the programs generated by the patterns, including a comparison with approaches that use a goal base to have declarative goals. An example of an issues that might be of particular interest in such comparison is how the use of plan patters will affect other aspects of agent-based development such as debugging. In the future, we also plan to support conjunctive goals such as $p \wedge q$ (where both p and q should be satisfied at the same time, as done in [19]), possibly through the use of plan patterns as well. Furthermore, we plan to investigate other patterns that may useful in the practical development of large-scale multi-agent systems.

Acknowledgements

Many thanks to A.C. Rocha Costa for discussions on maintenance goals in AgentSpeak. Anonymous reviewers for this paper have made detailed comments which helped improve the paper. Rafael Bordini gratefully acknowledges the support of The Nuffield Foundation (grant number NAL/01065/G).

References

1. D. Ancona and V. Mascardi. Coo-BDI: Extending the BDI model with cooperativity. In J. Leite, A. Omicini, L. Sterling, and P. Torroni, editors, *Declarative Agent Languages and Technologies, Proc. of the First Int. Workshop (DALT-03), held with AAMAS-03, 15 July, 2003, Melbourne, Australia*, number 2990 in LNAI, pages 109–134, Berlin, 2004. Springer-Verlag.
2. D. Ancona, V. Mascardi, J. F. Hübner, and R. H. Bordini. Coo-AgentSpeak: Cooperation in AgentSpeak through plan exchange. In N. R. Jennings, C. Sierra, L. Sonenberg, and M. Tambe, editors, *Proc. of the Third Int. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2004), New York, NY, 19–23 July*, pages 698–705, New York, NY, 2004. ACM Press.
3. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms, and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.
4. R. H. Bordini, J. F. Hübner, et al. ***Jason****: A Java-based AgentSpeak interpreter used with saci for multi-agent distribution over the net*, manual, release version 0.7 edition, Aug. 2005. `http://jason.sourceforge.net/`.
5. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the Golden Fleece of agent-oriented programming. In Bordini et al. [3], chapter 1.
6. R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226, Sept. 2004. Special Issue on Computational Logic in Multi-Agent Systems.
7. L. Braubach, A. Pokahr, W. Lamersdorf, and D. Moldt. Goal representation for BDI agent systems. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Second Int. Workshop on Programming Multiagent Systems: Languages and Tools (ProMAS 2004)*, pages 9–20, 2004.
8. P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(3):213–261, 1990.

9. M. Dastani, B. van Riemsdijk, F. Dignum, and J. Meyer. A programming language for cognitive agents: Goal directed 3APL. In *Proc. of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, volume 3067 of *LNAI*, pages 111–130, Berlin, 2004. Springer.
10. M. Dastani, M. B. van Riemsdijk, and J.-J. C. Meyer. Programming multi-agent systems in 3APL. In Bordini et al. [3], chapter 2.
11. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In M. P. Singh, A. S. Rao, and M. Wooldridge, editors, *Intelligent Agents IV—Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97), Providence, RI, 24–26 July, 1997*, number 1365 in LNAI, pages 155–176. Springer-Verlag, Berlin, 1998.
12. M. d'Inverno and M. Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):1–27, 1998.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents™ — summary of an agent infrastructure. In *Proceedings of Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS, held with the Fifth International Conference on Autonomous Agents (Agents 2001), 28 May – 1 June, Montreal, Canada*, 2001.
15. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [3], chapter 6.
16. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In Bordini et al. [3], chapter 6, pages 149–174.
17. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. Van de Velde and J. Perram, editors, *Proc. of the Seventh Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), 22–25 January, Eindhoven, The Netherlands*, number 1038 in LNAI, pages 42–55, London, 1996. Springer-Verlag.
18. A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.
19. B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Semantics of declarative goals in agent programming. In F. Dignum, V. Dignum, S. Koenig, S. Kraus, M. P. Singh, and M. Wooldridge, editors, *Proceedings of the 4rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 133–140. ACM, 2005.
20. M. B. van Riemsdijk, M. Dastani, and J.-J. C. Meyer. Subgoal semantics in agent programming. In C. Bento, A. Cardoso, and G. Dias, editors, *Proceedings of the 12th Portuguese Conference on Artificial Intelligence, EPIA 2005, Covilhã, Portugal, December 5-8, 2005*, volume 3808 of *LNCS*, pages 548–559, 2005.
21. M. Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. In Bordini et al. [3], chapter 7, pages 175–193.
22. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning*, 2002.

JADL - an Agent Description Language for Smart Agents

Thomas Konnerth, Benjamin Hirsch, and Sahin Albayrak

DAI Labor
Technische Universität Berlin
{Thomas.Konnerth,Benjamin.Hirsch,Sahin.Albayrak}@dai-labor.de

Abstract. In this paper, we describe the declarative agent programming language Jadl (JIAC Agent Description Language). Based on three-valued logic, it incorporates ontologies, FIPA-based speech acts, a (procedural) scripting part for (complex) actions, and allows to define protocols and service based communication. Rather than relying on a library of plans, the framework implementing Jadl allows agents to plan from first principles. We also describe the framework and some applications that have been implemented.

1 Introduction

The growth of interconnected devices, as well as the digitisation of content, has led to ever more complex applications running on ever more diverse devices. In recent years, the concept of service has become an important tool in coping with this development. Broadly speaking, services allow loosely coupled software entities to interact. Rather than providing a fixed and rigid set of interfaces, services provide means to adapt software to the ever faster changing environment of businesses. However, while the growing number of devices and networks poses a challenge to software engineering, it also opens the door to new application areas and offers possibilities to provide services on a new level of integration, context awareness, and interaction with the user.

In order to leverage the current and developing network and device technologies, a programming paradigm is needed that embraces distributed computing, open and dynamic environments, and autonomous behaviour.

Agent technology is such a paradigm. While there are many different areas and theories within the agent community, most work to make true the idea of an open, distributed, dynamic, and intelligent framework.

Without wanting to go into all the diverse subjects that research into agents encompasses, we want to point out some of the more prominent concepts and ideas here. On the level of single agents, BDI [1] has arguably been one of the most influential ideas. By assigning high level mentalistic notions to agents a new level of abstraction has been reached which allows to program agents in terms of goals rather than means. Agents contain not only functionality, but also the ability to plan (or alternatively a plan library) in order to achieve set goals. On the other hand, reactive behaviour is often desirable within agents, and should be supported in some way.

Research into interaction between agents is another important field. Here, agent communication languages attach semantic information about the "state of mind" of the sending agent [2]. Also, in order to enable interaction between agents, they need to understand each other. Ontologies allow agents to use a shared vocabulary, and to de-couple syntax and interpretation.

Agent-based technologies provide one possible and much sought-after approach to containing the complexity of today's soft- and hardware environment. However, while agents have been the subject of research for more than a decade, there are hardly any applications in the industry. There are differing views as to the reason for the slow uptake. Some blame a lack of "killer applications", or the general disconnect between research community and industry players. Others say that there is no problem at all, because industry uptake only happens at a certain maturity level has been reached [3]. Another reason that agent technologies have not been so successful is the lack of dedicated programming languages that allow the programmer to map agent concepts directly onto language constructs, and frameworks that cater for the needs of enterprise applications, such as security and accounting.

In this paper, we present the agent programming language Jadl (JIAC Agent Description Language). The thrust of the paper is to give a rather broad overview over the language — a planned series of papers will go into the different areas and cover them in greater detail.

The structure of this paper is as follows. After a broad overview over the different elements of Jadl (Section 2), we will describe its different features in some detail. In particular, we highlight knowledge representation in Section 3, followed by Section 4 with some words about programming the agents using reactive and planning elements. Section 5 finalises this part with a discussion on high-level communication. After introducing the framework that implements Jadl in Section 6, we proceed by presenting some of the projects that have been implemented using the framework (Section 7), and wrap up with some conclusions in Section 8.

2 Jadl Overview

Before we delve into different aspects of the language, it is important to give a broad overview over the language, in order to allow the reader to place the different elements of the language within their respective context.

Jadl is an agent programming language developed during the last few years at the DAI Laboratory of the Technische Universität Berlin. It is the core of an extensive agent framework called JIAC, and has originally been proposed by Sesseler [4]. As JIAC has been developed in cooperation with the telecommunications industry, it has until now not been available to the general public (though this might change soon, so watch this space!). Its stated goal is to support the creation of complex service-based applications. In Sections 6 and 7 we describe the framework and some exemplary implementations based on JIAC.

Jadl is based on three-valued predicate logic [5], thereby providing an open world semantics. It comprises four main elements: *plan elements*, *rules*, *ontologies*, and *services*.

While the first three elements are perhaps not too surprising, we should say a word or two about the last part, services. While we go into details in Section 5, we note here that agents communicate via services. From the perspective of the agent execution engine, a service call is handled the same way internal (complex) actions are executed. This is possible as services have the same structure as actions, having pre- and post conditions, as well as a body that contains the actual code to be executed. Reducing (or extending) communication to only consist of service calls allows us to incorporate advanced features like security and accounting into our framework. Also, programming communication becomes easier as all messages are handled in a clearly defined frame of reference.

Agents consist of a set of ontologies, rules, plan elements, and initial goal states, as well as a set of so-called AgentBeans (which are Java classes implementing certain interfaces). The state of the world is represented within a so-called fact base which contains instantiations of categories (which are defined in ontologies). AgentBeans contain methods which can be called directly from within Jadl, allowing the agent to interact with the real world, via user interfaces, database access, robot control, and more.

In the following sections, we will detail some different areas of Jadl, namely knowledge representation, agent behaviour, and communications.

3 Knowledge Representation

The language Jadl was designed to specifically meet the needs of open and dynamic agent systems. In a dynamic system where agents and services may come and go any time, the validity period of local information is quite short. Therefore, any system that allows and supports dynamic behaviour needs to address the issue of synchronisation and sharing of information. One answer to this is addressed by research in the area of transaction management (e.g. [6]). Our approach, however is to incorporate the idea of uncertainty about bits of information into our knowledge representation and thus allow the programmer to actively deal with outdated, incomplete or wrong data. Even leaving aside for a moment that there are unsolved issues when it comes to transaction management in multi-agent systems, we felt there are many cases when a real transaction-management would have been too much and it is quite acceptable and probably even more effective to just identify the bits of information that are inconsistent and afterwards update those bits.

We realised the concept of uncertainty by using a situation calculus that features a three valued logic. The use of logic allows us to use powerful and well known AI-techniques within a single agent. The third truth value is added for predicates that cannot be evaluated, with the information available to a particular agent. Thus, a predicate can be explicitly evaluated as *unknown.* This is an integral part of the language, and the programmer is forced to handle uncertainty when developing a new agent. Consequently, JIAC allows to handle incomplete or wrong information explicitly.

Jadl allows to define knowledge bases which are the basis of most of the rest of the language. Every object that the language refers to needs to be defined in an ontology. Jadl implements strong typing, i.e. contrary to for example Prolog, variables range over categories, rather than the full universe of discourse.

Categories are represented in a tree-like structure. Each node represents a category, with attached a set of (typed) attributes. Categories "inherit" attributes of ancestors.

Categories are specified as follows:

```
CatDecl = (cat CatName (ext CatName+) AttributeDecl*), where
AttributeDecl = (AttName Type Keyword*)
```

Keywords encode meta-information about the attributes.

To note here is that we allow multiple inheritance. Categories inherit *all* attributes of all ancestors. As attribute names are silently expanded to include the category structure, naming conflicts are avoided.

In addition to categories, Jadl allows to define functions and comparisons (which essentially are functions with a boolean, or rather 3-valued return type). The interpretation of functions is given by operational semantics. In practise, functions are encoded in Java.

While Jadl uses its own language to describe ontologies, we have developed a OWL-light to Jadl translator which allows JIAC agents to use published OWL-based ontologies.

Complex actions, or plan elements, describe the functional abilities of the agent. They in turn might call Java-methods, or use the Jadl scripting language. There are different types of plan elements — (internal) actions, and protocols and service invocations. All of them though have the same global structure. They consist of three main elements (in addition to the action name):

```
(act ActName pre PreCond eff Effect Body)
```

Pre-condition and Effect are described using logical formulae, consisting of elements defined in associated ontologies. It should be noted here that Jadl does not always allow the full power of first order formulae. For example, pre-conditions and effects can only consist of conjuncts. Also, formulae have to be written in disjunctive normal form. The body of an action can be either a script, a service, or an inference. Once the exectuion of this body is finished, the results are written to the variables, and afterwards the effect-formula is evaluated with these results to determine whether the action was successful. This way JIAC ensures that the actual result of an action does match the specified effect. Furthermore, protocols usually inherit the effect of their associated service. They may however have their own precondition, as there may be multiple protocols for a service - not all of which have to be applicable at a certain state.

4 Agent Behaviour

4.1 Goals and Action selection

As Jadl is meant to be interpreted in a BDI-like architecture, it includes the concept of achievement goals. These goals are implemented as simple formulae which an agent tries to fulfill once the goal is activated.

```
Goal = (goal Condition)
```

Once an agent has a goal, it tries to find an appropriate action that fulfills that goal. Such an action may either be a simple script or a service that is provided by another agent. For this selection, there is no difference between actions that can be executed locally and actions that are in fact services. The actual selection is done by comparing the formula stated in the goal (including the respective variable bindings) with the effects of all actions known to the agent. In this matching process, the literals of the formulae are compared, and if compatible, the values from the goal variables are bound to the corresponding variables of the action. After the action is completed, the results are writen to the original variables of the goal and the goal formula is evaluated to ensure that the goal is actually reached. If that is not the case, the agent is replanning its actions, and may try to reach the goal with other actions. One fact that should be mentioned here is that this matching of course considers the types of the variables. As these types may also include categories that come from ontologies, the matching process does also consider the semantic information that is present in those ontologies, e.g. inheritance.

4.2 Reactive Behaviour

Jadl allows to define *rules*. These rules are a means to realize the reactive behaviour of an angent. More specifically, a rule can give the agent a goal, whenever a certain event occurs. Rules are implemented in a rather straightforward fashion, consisting of a condition and two actions, one of which is executed when the condition becomes true, and the other when the condition becomes false.

```
Rule = (rule Condition Action Action)
```

Specifically, whenever an object is either added, removed, or changed in the fact base, the conditions that match the *object type* of the fact in question are tested against it, and execute the true or false action-part respectively. If the test yields unknown, no action is taken. The restriction of applicable rules to the matching object types is purely for efficiency purposes — if tested, rules whose condition does not match the fact will always yield `unknown`. Actions can themselves be either a new goal or a call to an AgentBean. In the former case, a new planning task for the agent is effectively created.

4.3 Planning

In the literature, there are numerous agent programming languages available. We can roughly classify them as logic based (such as AgentSpeak(L) [7, 8], 3APL [9, 10], Golog [11, 12], and MetateM [13, 14]) and Java based (such as Jack [15], Jade [16], Cougaar, [17] and MadKit [18]). The languages are mostly in the prototype stage, and provide high level concepts that implement some notion of BDI [19].

Generally, the concept of having beliefs, desires, and intentions, is "translated" into belief bases, goals, and a plan library. In particular, possibly with the exception of Golog and Cougaar, which allows for planning from first principles, all those languages assume a library of fully developed plans (modulo some parameters). A general execution cycle therefore maps internal and external states via some matching function to one or more plans, which are then (partially) executed.

While this approach certainly has its merits, in particular when it comes to execution speed, it is by no means clear that planning from first principles is not a viable alternative, certainly if approached with caution. The language we are presenting here has been used to implement numerous complex applications, showing that planning has its place and its uses in agent programming.

(Complex) Actions Before we detail the execution algorithm, we need to introduce the plan elements which are combined to plans which then are executed by the agent.

Plan elements can take a number of different forms. These include *actions*, as well as *protocols*, and *services*, which we will detail in the next subsection.

Actions, rather than being atomic elements, can be scripts. Jadl script provides keywords for sequential and parallel execution, conditionals, calls to AgentBeans, and even the creation of new goals, which then lead to new planning actions. It should be clear to the reader that extensive use of the scripting language, and especially the ability to trigger new plans, should be used with caution.

For a discussion on protocols and services, we refer the reader to Section 5.

While it is out of the scope of this paper to describe the action language in detail, we want to give the reader an impression in Figure 1. As Jadl is logic based, variables need to be bound and unbound to actual objects that are stored in the fact base. Also, formulae can be evaluated in order to ascertain their values. Sequential and parallel execution, as well as branching instructions can be used. Note further the keywords `iseq` and `seq` in the example. While the latter reflects a simple sequential execution of following elements, the former *iterates* through the given list (in this case a list of e-mail objects) and executes the sequence for each element. The `branch` statement executes the body if the test condition evaluates to true.

Plan Generation and Execution While Jadl can be used to provide a library of fully developed plans, its execution environment allows for planning from first principles. It employs the UCPOP algorithm [20], which generates a set of partial

```
(seq
    (unbind ?coredata)
    (unbind ?emailList)
    (unbind ?email)
    (eval (att coredata ?c ?coredata))
    (eval (att email ?coredata ?emailList))

    (bind ?haveIt false)

    (iseq ?emailList (var ?emailObj:EMailAddress)
        (seq
           (branch (isTrue (var ?haveIt))
             cont
             (par
               (eval (att email ?emailObj ?email))
               (bind ?haveIt true)
             )
             )
        )
    )
    (bind ?e ?email)
)
```

Fig. 1. Code Snippet of a complex script

plans based on a goal state and a set of actions. The partial plans are then "flattened" by a scheduler to create a full plan.

In order to create partial plans, the system first tries to reach the goal state by using local plan elements only, as this is considered the fastest and cheapest way of reaching a goal. If no plan can be found, the directory facilitator (DF) of the agent platform is contacted, and all available services are downloaded to the planning agent. Then, a second planning cycle is run, this time with the services registered at the DF included in the search. To limit the search space as far as possible, the algorithm ever only considers plan elements (and therefore services) that are relevant. Here, relevancy is determined by using ontology information on pre-conditions. So, a plan elements written for cars will be considered when looking for a BMW, but plan elements dealing with houses will not be used to expand the plan.

4.4 Scheduling and Failure Handling

In order to arrive at a full plan, the partial plans need to be ordered in a consistent fashion. As scheduling can be computationally expensive, the algorithm does little optimisation, and mainly ensures that the causal links (i.e. the order of actions that depend on each other) are met. Actions that are executed in parallel are not checked for consistency.

The actual execution has fall-back mechanisms on several levels. As can be seen in Figure 2, the execution of a goal (which can be either a single goal, or one of a number of steps that have been computed by the planner) is approached as follows. First, the locally known plan elements are matched against the goal. If one is found, and its pre-conditions are met, it is executed. If the preconditions are not yet fulfilled, the planner tries to find further planelements, that may meet the preconditions recursively. If either the goal or some preconditions cannot be met with the locally known planelements, a request is sent to the directory facilitator (DF), and the goal is again matched against the received set of services. As mentioned before, elements that are atomic actions for the planner (and execution model) can be complex actions, and even service calls. We will describe service calls in details later, and want to mention only that in the case of service calls, unsuccessful service invocations are also repeated with different service providers before re-initiating the process of finding a new action. Also note that the re-initialisation only occurs once, as otherwise a loop could occur.

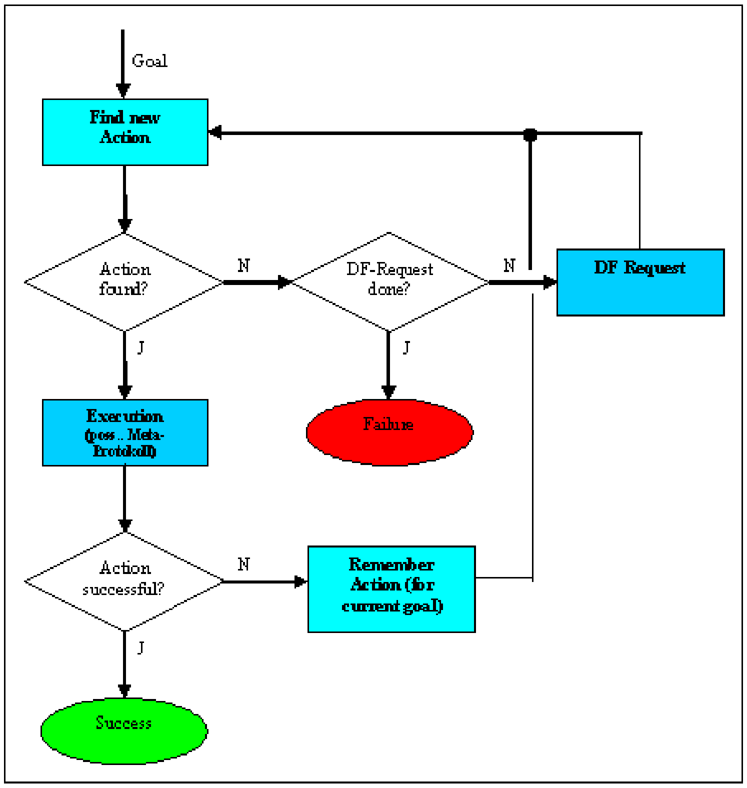


Fig. 2. Fulfilling a goal

5 High Level Communications

Protocols and services are used for communication purposes. In order to allow for an open system, agents solely communicate using service calls. Actual messages follow the FIPA ACL standard [21]. Rather than either exposing its whole functionality, or alternatively having an implicit representation of functionalities that might or might not be used by other agents, JIAC forces the programmer to define explicitly the functionalities that the agent exposes to the outside. This is done by explicitly configuring the list of services that are exposed to other agents. Each service has attached a number of protocols that can be executed during the service invocation, allowing for a conscious design of protocols. Figure 3 shows a small example which provides a time-synch service.

Figure 3 details a service definition. The example service is defined as an action (`act timeSyncService`) which has four elements. Firstly, a variable `?t` of type `TimeActualization` is declared. The type is defined in the `TimeSync` ontology. Second, we have the pre-conditions which must hold for the service to be executed. In our example, this is set to true, but can be any conjunctive formula (and can include *unkown* attributes as well. Thirdly, the effect of executing the service is described. The example service sets the attribute `locallySynchronized` of the object assigned to `?t` to `true`. Finally the actual service description starts.

A service consists of a service object, which is defined by a name, a set of protocols, and some ontologies. We should note here that the set of protocols includes protocols for negotiation as well as service provision. Figure 3 for example defines two protocols. The first describes the actual service protocol which implements the body of the service, while the `contractNet` protocol has the flag `multi true` which defines it as a one-to-many negotiation protocol that is used for provider-selection.

The actual linking of protocols to services happens during runtime. Whenever an agent decides to execute a service it looks up the corresponding protocols (which are identified by their names) an tries to negotiate the protocol with the service-partner. If they can find a common protocol, both protocol-sides are intiated, otherwise the service fails.

Channelling communication through services makes security much easier to implement. This is because agents can only interact through the clearly defined service invocation, rather than any sort of interaction. Secondly, services can define additional meta-data such as costs, AAA, or QoS in a clear and consistent manner, allowing agents (and their owners) to have clear policies concerning the provision of functionalities to third party contacts. Again, allowing for simple message exchanges makes accounting very complex.

The last two points, security and accounting, are important aspects of any industrial application of agent technology. Only if we can guarantee a certain level of security, and only if we can ensure that services offered can actually be accounted according to clear and definable policies can we ever hope to convince industry players to consider agent technology as a viable alternative to today's technologies.

```
(act timeSyncService
   (var ?t:TimeActualization)
   (pre true)
   (eff
     (att locallySynchronized ?t true)
   )
   (service
     (obj Service:DAI_1
       (name "timeSyncService")
       (protocols
         [Protocol:
           (obj Protocol
             (name "timeSyncServiceProtocol")
             (provider true)
           )
           (obj Protocol
             (name "contractNet")
             (multi true)
           )
         ]
       )
       (ontologies
         {string:
         "de.dailab.jiac.ontology.Service:DAI_1"
         "de.dailab.scb.ontology.TimeSync:DAI_1"
         }
       )
     )
   )
 )
```

Fig. 3. Example of a service definition in Jadl

Meta-Protocol

As mentioned before, service calls are wrapped by a meta-protocol in JIAC (see Figure 4), which deals with session handling, security, accounting, provider and transport selection, and error handling, leaving the programmer to concentrate on the actual functionality and protocol interaction.

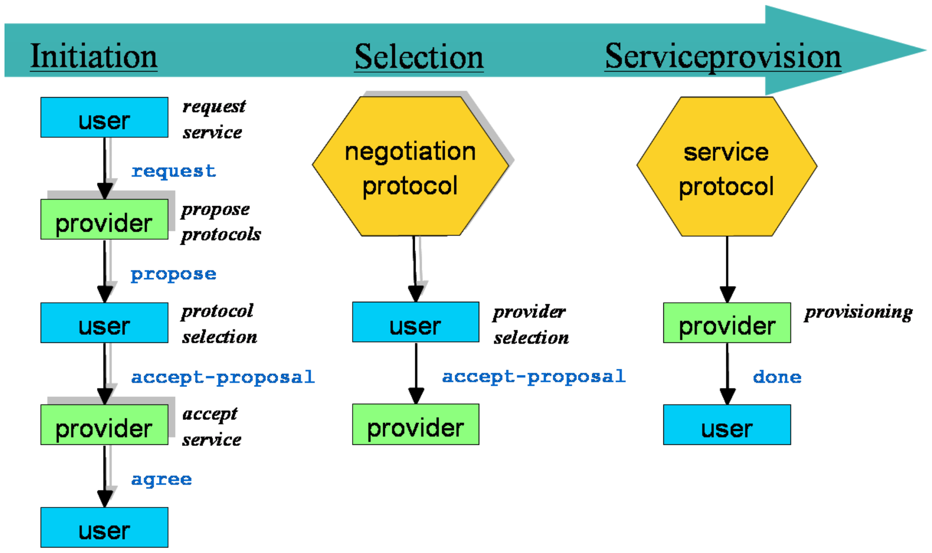


Fig. 4. Graphical representation of the meta-protocol

In order to trigger a service invocation, the agent must have failed to satisfy a goal which using just actions that are available by the agent itself. This includes services, that the agents provides by himself. If such a situation occurs, the agent sends a request to the DF, which answers with a list of services that could fulfil the goal. The agent then chooses one service, and notifies the DF, which again sends back a list, but this time of agents that are providing the requested service.

As Figure 4 shows, a service invocation consists of three distinct phases. During the initiation phase, the user and provider(s) agree on a protocol to use. This includes security negotiations, as well as accounting and QoS requirements.

Once this is done, a (optional) negotiation protocol is triggered, during which the actual provider agent is chosen. Only then, the actual service is invoked.

To note here is that while a service provision is always a one-to-one communication, the actual service selection allows for one-to-many communication. If the negotiation protocol is empty, the first service provider is chosen. The meta protocol catches any errors that might occur during service provision (i.e. time outs, or cancel- and not-understood messages), and reacts accordingly. For example, in case of a failed service provisioning, it returns to the selection phase and chooses another agent that can provide the service. Only once no more agents are available does the service provisioning fail (from the point of view of the agent). In that case, a re-plan action is triggered.

For a more detailed description of the meta-protocol we refer the reader to [22].

6 JIAC

In the preceding sections we have described the Jadl language. Now, we describe the JIAC framework which implements Jadl.

JIAC consists of a (java-based) run-time environment, a methodology, tools that support the creation of agents, as well as numerous extensions, such as web-service-connectivity, accounting and management components, device independent interaction, an owl-to-Jadl translator, a OSGI-connector and more. An agent consists of a set of application specific java-classes, rules, plan elements, and ontologies. Strong migration is supported, i.e. agents can migrate from one platform to another during run-time. JIAC's component model allows to exchange, add, and remove components during run time. Standard components (which themselves can be exchanged as well) include a fact-base component, execution-component, rule-component, and more. A JIAC agent is defined within a property file which describes all elements that the agent consists of.

JIAC is the only agent-framework that has been awarded a common criteria EAL3 certificate, an internationally accepted and renowned security certificate.

Conceptually, an agent system consists of a number of platforms, each of which has its own directory facilitator and agent management system. The DF registers the agents on the platform, as well as the services that they offer. We have investigated a number of different techniques to connect different DF's, such as P2P and hierarchical approaches. On each platform, a number of agent "lives" at each moment. Agents themselves implement one or more agent roles. Each role consists of the components that are necessary to implement it. Usually, this will include plan elements, ontologies, rules, and AgentBeans. Here, Jadl and the elements that can be described using it come into play.

Currently we finalise a new version of the tool-suite which is based on Eclipse. Programming in Jadl, as well as creating and running JIAC agents is supported on different levels. Additional to text-based support elements such as syntax highlighting and code folding, most Jadl elements can be displayed and edited in a graphical interface, removing the sometimes awkward syntax as far as possible from the user, and allowing her to focus on functionality rather than syntax debugging.

In addition to the tools that support Jadl itself, we have created a number of additional tools. A security tool provides methods to manage certificates, and ensure secure communication between agents. An accounting tool provides the user with means to create and manage user databases and related elements such as tariff information. The Agent configurator allows to display and modify agent's components during run-time, and to change goals, plan elements, and so forth. We have also incorporated advanced testing and logging features, to facilitate debugging and the general quality of the produced code. Without wanting to go into details, we have extended the Unit-test approach to agents, thereby providing a test-environment where interactions of agents can be tested automatically, for example in conjunction with a cruise-control server.

While tools help to hide the inherent complexity, they can only partially support the programmer during the design phase of a project. Recognising this, JIAC provides users with a methodology which is rooted in the concepts of Jadl, and of JIAC. Here, we focus not only on design but also on practical needs of project management. The JIAC methodology describes the interaction between customer, designer, and project manager, and uses the agile programming approach [23]. Continuous integration is another important element of the methodology, and is supported by above described testing environment.

Both, the methodology and the tool-suite support re-use of components. On the tool side, we are currently implementing a repository which can can be accessed via the network, and which holds functionality that can be included in projects. On the methodology side, special care is taken to enable re-use during analysis, design, and implementation. It also encourages programmers to refine new functionality to a re-usable form towards the end of the project, facilitating further the re-use of components.

Most importantly, the service concept supports re-use of functionality by design. Each created service can be invoked by other agents, thereby offering the most natural re-use of functionality.

Another extension to JIAC is the IMASU (Intelligent Multi-Access Service Unit). With it, interfaces between agents and (human) users can be described abstractly. The unit creates an appropriate user interface for a number of devices, such as web-browsers (HTML), PDA's and mobile phones (WML), and telephone (VoiceXML) [24].

7 Implemented Applications

To give the reader an idea about the power of the framework, we present some of the projects that have been implemented.

BerlinTainment This project is aimed at simplifying the provision of information over the internet. In order to provide cultural and leisure related functionality to visitors of Berlin, a personalised service based on the JIAC framework has been developed. Agents provide and integrate information from restaurants, route planners, public transport information, cinemas, theatres, and more. Using BerlinTainment, users can plan their day out, make reservations, be guided to the various locations, and be informed about touristic sites from one place, and with various devices [25].

PIA (Personal Information Agent) concerns the collection, dissemination, and provision of personalised content. It employs agents on three layers. Firstly, extractor agents monitor sources of information and extract content provided in different formats, such as HTML pages, PDF, and Microsoft Word documents. Secondly, filter-agents analyse the content based on preferences of the users. Thirdly, presentation agents control the presentation and output of the filtered data, again based on the users preference and device. PIA is used internally in our institute to collect information concerning research projects and grants, as well as providing personalised news-letters [26].

8 Conclusion

In this paper we have presented the agent programming language Jadl. Based on three-valued logic, it provides constructs to describe ontologies, protocols and services, and complex actions. JIAC agents use a planner to construct plans from those actions. There, internal actions and service invocations are handled transparently to the planning component.

The Jadl language and its framework, JIAC, provide arguably all elements that are needed for a successful agent deployment. JIAC provides tools, a methodology, and a host of extensions that provide extensions like webservice-interaction, OSGI-connectors, accounting, security, and network components that support the creation of complex services in commercial settings.

We do not claim to have created a language that the best choice for creating anything related with agents. However, we have tried to show that Jadl covers a host of issues that we think should be covered by agent programming languages. Using JIAC, several large implementations have been done, and shown to us the merits of the language.

9 Acknowledgements

Jadl and JIAC are based on work done by Sesseler [4] in the course of his doctoral thesis. JIAC has been developed with the kind support of Deutsche Telekom.

References

1. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., Sandewall, E., eds.: Principles of Knowledge Representation and Reasoning: Proc. of the Second International Conference (KR'91). Morgan Kaufmann, San Mateo, CA (1991) 473–484
2. Labrou, Y., Finin, T., Peng, Y.: The current landscape of agent communication languages. IEEE Intelligent Systems **14** (1999) 45–52
3. Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent based computing - agent technology roadmap. Roadmap, AgentLink III (2005) Draft Version of July 2005.
4. Sesseler, R.: Eine modulare Architektur für dienstbasierte Interaktion zwischen Agenten. Doctocal thesis, Technische Universität Berlin (2002)
5. Kleene, S.C.: Introduction to Metamathematics. Wolters-Noordhoff Publishing and North-Holland Publishing Company (1971) Written in 1953.

6. Kotagiri, R., Bailey, J., Busetta, P.: Transaction oriented computational models for multi-agent systems. In: Proc. 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2001), IEEE Press (2001) 11–17
7. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In van Hoe, R., ed.: Agents Breaking Away, 7^{th} European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96,. Volume 1038 of Lecture Notes in Computer Science., Eindhoven, The Netherlands, Springer Verlag (1996) 42–55
8. Bordini, R.H., Hübner, J.F., et al.: Jason: a Java Based AgentSpeak Interpreter Used with SACI for Multi-Agent Distribution over the Net. 5^{th} edn. (2004)
9. Dastani, M.: 3APL Platform. Utrecht University. (2004)
10. Hindriks, K.V., Boer, F.S.D., der Hoek, W.V., Meyer, J.J.: Agent programming in 3apl. Autonomous Agents and Multi-Agent Systems **2** (1999) 357–401
11. Giacomo, G., Lesperance, Y., Levesque, H.: Congolog, a concurrent programming language based on the situation calculus: Foundations. Technical report, University of Toronto (1999)
12. Giacomo, G., Lesperance, Y., Levesque, H.: Congolog, a concurrent programming language based on the situation calculus: Language and implementation. Technical report, University of Toronto (1998)
13. Finger, M., Fisher, M., Owens, R.: Metatem at work: Modelling reactive systems using executable temporal logic. In: Proceedings of the International Conference on Industrial and Engeneering Applications of Artificial Intelligence, Gordon and Breach (1993)
14. Fisher, M., Ghidini, C., Hirsch, B.: Programming groups of rational agents. In Dix, J., Leite, J., eds.: CLIMA IV, Fourth International Workshop. Volume 2359 of LNAI. (2004) 16–33
15. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK — components for intelligent agents in java. Technical report, Agent Oriented Software Pty, Ltd. (1999)
16. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - a FIPA-compliant agent framework. Internal technical report, CSELT (1999) Part of this report has been also published in Proceedings of PAAM'99, London, April 1999, pp.97-108.
17. Helsinger, A., Thome, M., Wright, T.: Cougaar: A scalabe, distributed multi-agent architecture. In: IEEE SMC04. (2004)
18. Gutknecht, O., Ferber, J.: The MADKIT agent platform architecture. Technical Report R.R.LIRMM00xx, Laboratoire d'Informatique, de Robotqiue et de Microélectronique de Montpellier (2000)
19. Bratman, M.E.: Intentions, Plans, and Practical Reason. Havard University Press, Cambridge, MA (1987)
20. Penberthy, J.S., Weld, D.: UCPOP: A sound, complete, partial-order planner for ADL. In: Proceedings of Knowledge Review 92, Cambridge, MA (1992) 103–114
21. FIPA: Fipa acl message structure specification (2002)
22. Albayrak, S., Konnerth, T., Hirsch, B.: Ensuring security and accountability in agent communication. In Preparation (2005)
23. Lyons, K.: The agile approach. Technical report, Conoco Phillips Australia Pty Ltd. (2004)
24. Rieger, A., Cissée, R., Feuerstack, S., Wohltorf, J., Albayrak, S.: An agent-based architecture for ubiquitous multitmodal user interfaces. In: The 2005 International Conference in Active Media Technology. (2005)

25. Wohltorf, J., Cissée, R., Rieger, A.: BerlinTainment: An agent-based context-aware entertainment planning system. IEEE Communications Magazine **43** (2005) 102–109
26. Albayrak, S., Dragan, M.: Generic intelligent personal information agent. In: International Conference on Advances in Internet, Processing, Systems, and Interdisciplinary Research. (2004)

Agreeing on Defeasible Commitments

Ioan Alfred Letia[1] and Adrian Groza[1]

Technical University of Cluj-Napoca
Department of Computer Science
Baritiu 28, RO-3400 Cluj-Napoca, Romania
{letia,adrian}@cs-gw.utcluj.ro

Abstract. Social commitments are developed for multi-agent systems according to the current practice in law regarding contract formation and breach. Deafeasible commitments are used to provide a useful link between multi-agent systems and legal doctrines. The proposed model makes the commitments more expressive relative to contract law, improving the model for the life cycle of the commitments. As a consequence, the broader semantics helps in modelling different types of contracts: gratuitous promises, unilateral contracts, bilateral contracts, and forward contracts. The semantics of higher-order commitments is useful in deciding whether to sign an agreement or not, due to a larger variety of protocols and contracts.

1 Introduction

Artificial agents and the contracts they make are ubiquitous, while at the same time, there is a lack of application of the current practice in law to multi-agent systems (MAS). From the point of view of law, there is a philosophical debate regarding when to attach person-hood to artificial agents. The actual context of web services representing business entities and agents interacting with services implies legal responsibilities for each agent. From the engineering point of view, agents have to be built and synchronized with the norms and values of society.

Social commitments were introduced as a way to capture the public aspects of communications [1] and research has been focused on the development of agent communication languages and flexible interaction protocols [2, 3]. As commitments appear to be sometimes too restrictive (direct obligations) and sometimes too flexible, allowing unconstrained modification of commitments, social commitments should be more flexible than usual obligations but also more constrained than permissions [1]. On this line, we apply principles of contract law as an objective measure to decide on the flexibility of the operations on commitments, beginning with a commitment-based representation of different types of agreements from contract law. The main advantage of applying current practice in law to model commitments within multi-agents systems is that the principles of contract law are verified and polished during years of economical and judicial practice.

Modeling agent communication implies several approaches: mental (BDI and modalities), social (which highlights the public and observable elements like social commitments that agents exchange when conversing), and argumentative (based on agent reasoning capabilities). When participating in an agreement, agents should use their mental states, share information and reason about new facts. We seek to synchronize the social commitments developed for MAS with the existing legal doctrines, which the law applies in case of contract formation. We define a framework by using the temporalised normative positions in defeasible logic [4] to introduce defeasible commitments for representing contract laws [5] in the model of the life cycle of commitments.

2 Temporalised normative positions

For defining defeasible commitments, we are using the temporalised normative positions [4]. A theory in *normative defeasible logic* (NDL) is a structure (F, R_K, R_I, R_A, R_O, $\succ$) where F is a finite set of facts, R_K R_I R_A R_O are respectively a finite set of persistent or transitive rules (strict, defeasible, and defeaters) for knowledge, intentions, actions, and obligations, and $\succ$ representing the superiority relation over the set of rules.

A rule in NDL is characterized by three orthogonal attributes: modality, persistence, strength. As for modality, R_K represents the agent's theory of the world, R_A encodes its actions, R_O the normative system or his obligations, while R_I and the superiority relation capture the agent's strategy or its policy. *A persistent rule* is a rule whose conclusion holds at all instants of time after the conclusion has been derived, unless a more powerful rule, according to the superiority relation, has derived the opposite conclusion. *A transient rule* establishes the conclusion only for a specific instance of time [4].

Strict rules are rules in the classical sense, that is whenever the premises are indisputable, then so is the conclusion, while *defeasible rules* are rules that can be defeated by contrary evidence. For "sending the goods means the goods were delivered", if we know that the goods were sent then they reach the destination, unless there is other, not inferior, rule suggesting the contrary. *Defeaters* are rules that cannot be used to draw any conclusions. Their only use is to prevent some conclusions, as in "if the customer is a regular one and he has a short delay for paying, we might not ask for penalties". This rule cannot be used to support a "not penalty" conclusion, but it can prevent the derivation of the penalty conclusion.

$\rightarrow^t_X$, $\Rightarrow^t_X$ and $\rightsquigarrow^t_X$ denote transitive rules (strict, defeasible, defeaters), while $\rightarrow^p_X$, $\Rightarrow^p_X$ and $\rightsquigarrow^p_X$ denote persistent rules (strict, defeasible, defeaters), where $X \in \{K, I, A, O\}$ represents the modality. A conclusion in NDL is a tagged literal where $+\Delta^\tau_X q{:}t$ means that q is definitely provable of modality X, at time t in NDL (figure 1); and $+\partial^\tau_X q{:}t$ means that q is defeasibly provable of modality X, at time t in NDL (figures 2, 3). Here $\tau \in \{t, p\}$, t stands for transient, while p for a persistent derivation. A strict rule $r \in R_s$ is $\Delta_X - applicable$ if $r \in R_{s,X} \forall a : t_k \in A(r) : a_k : t_k$ is $\Delta_X - provable$. A strict rule $r \in R_s$ is

$\Delta_X - discarded$ if $r \in R_{s,X} \exists a_k : t_k \in A(r) : a_k : t_k$ is $\Delta_X - rejected$, and similarly for ∂. The conditions for concluding whether a query is transient or

$+\Delta^t_X$: If $P(i+1) = +\Delta^t_X q : t$ then
  $q : t \in F$, or
  $\exists r \in R^t_{s,X}[q : t]$ r is $\Delta_X - applicable$

$+\Delta^p_X$: If $P(i+1) = +\Delta^p_X q : t$ then
  $q : t \in F$, or
  $\exists r \in R^p_{s,X}[q : t]$ r is $\Delta_X - applicable$ or
  $\exists t' \in \Gamma : t' < t$ and $+\Delta^p_X q : t' \in P(1..i)$.

Fig. 1. Transient and persistent definite proof for modality X

persistent, definitely provable is shown inthe figure 1. For the transient case, at step $i+1$ one can assert that q is definitely transient provable if there is a strict transient rule $r \in R^t_s$ with the consequent q and all the antecedents of r have been asserted to be definitely (transient or persistent) provable, in previous steps. For the persistent case, the persistence condition allows us to reiterate literals definitely proved at previous times. For showing that q is not persistent definitely provable, in addition to the condition we have for the transient case, we have to assure that, for all instances of time before now the persistent property has not been proved. According to the above conditions, in order to prove that q is definitely provable at time t we have to show that q is either transient, or persistent definitely provable [4].

$+\partial^t_X$: If $P(i+1) = +\partial^t_X q : t$ then
  (1) $+\Delta_X q : t \in P(1..i)$ or
  (2)$-\Delta_X \sim q : t \in P(1..i)$ and
    (2.1) $\exists r \in R_{sd,X}[q : t]$: r is ∂_X-applicable and
    (2.2) $\forall s \in R[\sim q : t]$: s is ∂_X-discarded or
      $\exists w \in R(q : t)$: w is ∂_X-applicable or $w \succ s$

Fig. 2. Transient defeasible proof for modality X

Defeasible derivations have an argumentation like structure [4]: firstly, we choose a supported rule having the conclusions q we want to prove, secondly we consider all the possible counterarguments against q, and finally we rebut all the above counterarguments showing that, either some of their premises do not hold, or the rule used for its derivation is weaker than the rule supporting the initial conclusion q. A goal q which is not definitely provable is defeasibly transient provable if we can find a strict or defeasible transient rule for which

```
$+\partial_X^p$: If $P(i+1) = +\partial_X^t q : t$ then
    (1) $+\Delta_X^p q : t \in P(1..i)$ or
    (2)$-\Delta_X \sim q : t \in P(1..i)$, and
        (2.1) $\exists r \in R_{sd,X}^p[q : t]$: $r$ is $\partial_X$-applicable, and
        (2.2) $\forall s \in R[\sim q : t]$: either $s$ is $\partial_X$-discarded or
            $\exists w \in R(q : t)$: $w$ is $\partial_X$-applicable or $w \succ s$; or
    (3) $\exists t' \in \Gamma : t' < t$ and $+\partial_X^p q : t' \in P(1..i)$ and
        (3.1) $\forall s \in R[\sim q : t"], t' < t" \leq t$, $s$ is $\partial_X$-discarded, or
            $\exists w \in R(q : t")$: $w$ is $\partial_X$-applicable and $w \succ s$.
```

Fig. 3. Persistent defeasible proof for modality X

all its antecedents are defeasibly provable, $\sim q$ is not definitely provable and for each rule having $\sim q$ as a consequent we can find an antecedent which does not satisfy the defeasible provable condition (figure 2). For the persistence case, the aditional clause (3) from figure 3 verifies if the literal $q : t$ has been persistent defeasibly proved before, and this conclusion remained valid all this time (there was no time t" when the contrary $\sim q$ was proved by firing the rule s, or the respective rule was no stronger than the one sustaining q).

3 Types of commitments

The classical definition of a conditional commitment states that a commitment is a promise from a debtor x to a creditor y to bring about a particular sentence p under a condition q. Starting from this definition we provide a generalized commitment abstract data type.

Definition 1. *A commitment is a relation*

$$C_m^n(x, y, q^n : [t_{issue}], [\star]p^m : [t_{maturity}]) : [t_{expiration}]$$

with optional literals within square brackets, representing the promise p made by debtor x to creditor y in exchange of which the action q is requested, where the time of maturity $t_{maturity}$ shows the time remaining until the promise p^m is satisfied by the debtor x if the request q^n holds until time t_{issue} and $\star \in \{+\Delta, -\Delta, +\partial, -\partial, ?\}$ is an optional tag used to express informing messages.

The parameters m and n help us to define meta commitments or higher-order commitments. Their role is to provide a rich semantics used to express a large variety of contractual clauses or negotiation patterns: m is a measure of the promises made by the debtor, while n is a measure of the requests made by the debtor (figure 4). We define two operators for the composition of commitments: $\circ_q$ which deals with requests and $\circ_p$ which deals with promises.

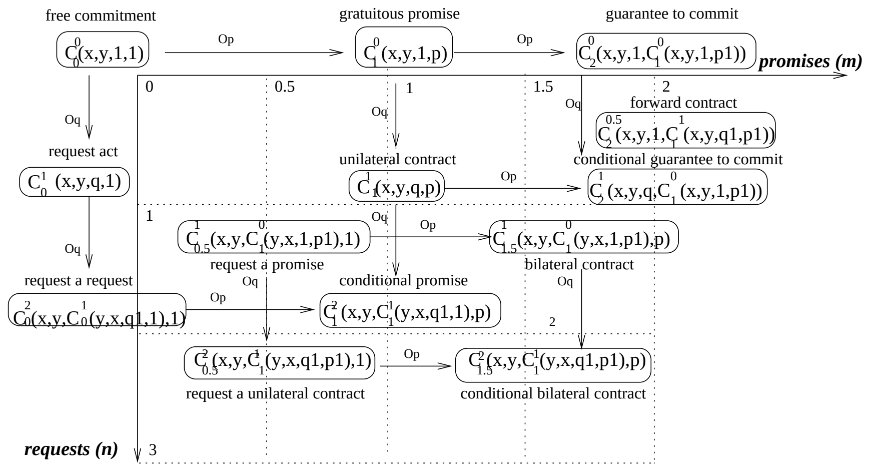


Fig. 4. Types and composition of the commitments

3.1 Contractual commitments

When $m \in [1, 2)$ we name the resulting commitments contractual commitments. Next, we discuss each type of contractual commitments from a legal point of view.

The example "I will give you the item g_1 in 5 days." is represented by $C_1^0(me, you, 1, g_1 : 5)$, defined by law as *gratuitous promise.*

Definition 2. *In a Gratuitous Promise (n=0, m=1) the debtor x promises the creditor y to bring about p until $t_{maturity}$ without requesting anything ($n = 0$).*

$$C_1^0(x, y, 1, p_1 : t_{maturity})$$

The example "I will give you the item g_1 in 5 days after you will pay the price" will be represented by $C_1^1(me, you, pay(you) : t_{pay}, g_1 : t_{pay} + 5)$, and "I will give you the item g_1 as long as the oil price is 135\$" by $C_1^1(me, you, price = 135 : t_{price}, g_1 : t_{price} + 5)$. In the first example the condition is brought about by the creditor y, while in the second the condition is an environment fact and does not necessarily depend on y. The law defines such a commitment a *unilateral contract*, involving an exchange of the offerer's promise (p) for the oferee's act (q), with the completion of the act required to indicate acceptance.

Definition 3. *A Unilateral Contract (n=1, m=1) involves an exchange of the offerer's promise p for the oferee's act q, where the debtor x promises the creditor y to bring about p until $t_{maturity}$ if condition q holds at time t_{issue}.*

$$C_1^1(x, y, q : t_{issue}, p : t_{maturity})$$

Consider the examples "I will give you the item g_1 no later than 5 days, if you promise me in maximum 1 day that you will pay the price no later than 3 days" represented as $C^1_{1.5}(me, you, C^0_1(you, me, 1, pay : 3) : 1, g_1 : 5)$ and "I will give you the item g_1 no later than 5 days, if the bank promises me in maximum one day to pay the price no later than 3 days" as $C^1_{1.5}(me, you, C^0_1(bank, me, 1, pay : 3) : 1, g_1 : 5)$. According to contract law, a contract in which both sides make promises is called a *bilateral contract.*

Definition 4. *In a Bilateral Contract (n=1, m=1.5) both sides make promises, the debtor x promises the creditor y to bring about p if the creditor y promises x to bring about p_1.*

$$C^1_{1.5}(x, y, C^0_1(y, x, 1, p_1), p)$$

We note that a $C^1_{1.5}$ commitment is somehow weaker than a C^1_1 commitment. This fine grained mechanism opens the possibility of designing agents with different levels of attitude towards risk and it also refines the idea of leveled commitment contracts [6].

"I will give you the item g_1 no later than 5 days, if you promise me to pay the price no later than 3 days under the condition that oil price reaches 135$; my offer expires in 10 days." is represented by $C^2_{1.5}(me, you, C^1_1(you, me, oilPrice = 135, pay : 3) : 10, g_1 : 5)$.

Definition 5. *In a Conditional Bilateral Contract (n=2, m=1.5) the debtor x promises the creditor y to bring about p if agent y promises x to bring about p_1 under condition q_1.*

$$C^2_{1.5}(x, y, C^1_1(y, x, q_1, p_1), p)$$

Here $n = 2$ means that agent x has two requests: it requests the promise C^1_1 which contains the second request q_1. On the other hand, $m = 1.5$ means that it promises p and also p_1 which, being an inner promise, in our model weighs only 0.5. The above semantics includes a form of negotiation because, at the creation of the inner commitment, both $C^2_{1.5}$ and C^1_1 commitments are open offers (see section 4). Therefore, the agents are not committed to them and they may be canceled anytime in this state, without considering it a breach.

3.2 Request commitments

When $m \in [0, 1)$ the debtor does not promise anything directly, called request commitments. For both $m = 0$ and $n = 0$ we have a *free commitment* $C^0_0(x, y, 1, 1)$, while $n \neq 0$ gives the following types of requests.

"Please pay me the price of the product g_1 in two days" is represented as a *request act* $C^1_0(me, you, price : 2, 1)$[1].

Definition 6. *In a Request Act (n=1, m=0) the debtor x requests the creditor y to bring about q until time t_{issue}.*

$$C^1_0(x, y, q : t_{issue}, 1)$$

[1] With $n = 1$ we denote $q^1 = q$ and $p^0 = 1$.

Observe that the debtor does not promise anything. The acceptance of the above request is made simply by causing the sentence q or performing the requested action. If the requested act is a negative sentence, it represents a *taboo* [7] or interdiction.

"Please promise me that you will pay for the item in 3 days" is represented as $C^1_{0.5}(me, you, C^0_1(you, me, 1, pay : 3), 1)$.

Definition 7. *A Request a Promise (n=1, m=0.5) is used by a debtor x to request the creditor y to promise until* $t_{expiration}$ *that it will bring about* p_1 *until* $t_{maturity}$

$$C^1_{0.5}(x, y, C^0_1(y, x, 1, p_1 : t_{maturity}) : t_{expiration}, 1)$$

obtainable from $C^1_0 \circ_q C^0_1$.

The acceptance of the request is done by creating the inner commitment $C^0_1(y, x, 1, p_1 : t_{maturity})$ until the deadline $t_{expiration}$. When the time-out elapses the request commitment reaches the failed state. If the creditor wants to explicitly reject the request, it will respond by creating the negative commitment $\neg C^0_1(y, x, 1, pay : 3) : 5$, having the same deadline with the request commitment[2]. The meaning of the above rejection is "I will not commit to you to bring about p_1 in 3 days; I will reconsider your request after 5 days".

"Ask me to give you the money" is shown as $C^2_0(me, you, C^1_0(you, me, money, 1), 1)$ and "Please request the bank to pay you" as $C^2_0(me, you, C^1_0(you, bank, pay, 1), 1)$.

Definition 8. *In a Request a Request (n=2, m=0) the debtor x requests the creditor y to request the sentence* q_1 *from another agent* z[3] *until time* t_e

$$C^2_0(x, y, C^0_1(y, z, q_1, 1) : t_{expiration}, 1)$$

obtainable from $C^0_1 \circ_q C^0_1$.

"Please buy me shares as soon as their price reaches 10\$" is represented by $C^2_{0.5}(me, you, C^1_1(you, me, price = 10, buy), 1)$.

Definition 9. *In a Request a Unilateral Contract (n=2, m=0.5) the debtor x requests the creditor y to commit to bring about* p_1 *if the condition* q_1 *holds*

$$C^2_{0.5}(x, y, C^1_1(y, z, q_1, p_1) : t_{expiration}, 1)$$

obtainable from $C^0_1 \circ_q C^1_1$.

3.3 Guarantee commitments

In these commitments the debtor promises that a specific commitment will exist in a given window of time.

For "I guarantee you that the bank will commit in maximum 7 days to give you the credit" we use the formula $C^0_1(me, you, 1, C^0_1(bank, you, 1, credit) : 7)$.

[2] Otherwise a form of negociation may arise.

[3] The agent z may be the debtor x.

Definition 10. *In a Guarantee to Commit (n=0, m=2) the debtor x guarantees the creditor y that a special commitment will exist until* $t_{expiration}$

$$C_2^0(x, y, 1, C_1^0(z, y, 1, p_1) : t_{expiration})$$

obtainable from $C_1^0 \circ_p C_1^0$.

If $z = y$ the creditor manifests its own intention to commit or it guarantees that it will make the respective gratuitous promise no longer than $t_{expiration}$. It can be seen as a precommitment or an intention to commit.

”If you have all the papers, I promise you that the bank will commit in maximum 7 days to give you the credit” is represented as $C_2^1(me, you, papers, C_1^0(bank, you, 1, credit) : 7))$.

Definition 11. *In a Conditional Guarantee to Commit (n=1, m=2) the debtor x guarantees the creditor y that a specific commitment will exist until* $t_{expiration}$ *if condition q holds*

$$C_2^1(x, y, q, C_1^0(z, y, 1, p_1) : t_{expiration})$$

obtainable from $C_1^1 \circ_p C_1^0$.

We represent ”I commit you to sell my house to you next year at the price 20000\$” by $C_2^{0.5}(me, you, 1, C_1^1(me, you, 20000, house) : 365))$.

Definition 12. *In a Forward Unilateral Contract (n=0.5, m=2) the debtor x guarantees the creditor y that a specific unilateral contract will exist until* $t_{expiration}$.

$$C_2^{0.5}(x, y, 1, C_1^1(z, y, q_1, p_1) : t_{expiration})$$

According to contract law, the particular case in which $z = x$ is a form of a *forward contract*, obtainable from $C_1^0 \circ_p C_1^1$. Applying the composition operators $\circ_q$ or $\circ_p$ we can also model *forward bilateral contracts* and *forward conditional bilateral contracts.*

3.4 Informing commitments

We see the informing act as a form of commitment in the sense that the agent who propagates some information guarantees its validity. In other words, it is committed to the creditor that the notified fact is true, based on the debtor's view of the world. Contract law names such type of statement *terms*. The truth of the term is guaranteed by the agent that made the statement. We use this type of commitment to allow information sharing between agents. The literature shows that information sharing is a key-point in the coordination of multi-agent systems.

The situation ”My partner informs me that he has already sent the money, while the bank says that the payment has not been made yet” is coded with $C_1^0(partner, me, 1, +\partial_K^p pay)$ and $C_1^0(bank, me, 1, -\partial_K^p pay)$. The agent me will fire both defeasible rules $r_1 : C_1^0(partner, me, 1, +\partial_K^p pay) \Rightarrow pay$ and $r_2 : C_1^0(bank, me, 1, -\partial_K^p pay) \Rightarrow \neg pay$, but it will give more credit to the statement of the bank $r_2 > r_1$.

Definition 13. *In a Fact Notification the debtor x informs creditor y if a specific sentence p is $+\Delta^{\tau}_{X}p$, $-\Delta^{\tau}_{X}p$, $+\partial^{\tau}_{X}p$, or $-\partial^{\tau}_{X}p$ according to its defeasible theory D.*

$$C_1^0(x, y, 1, \star p)$$

"I inform you that agent z has an active commitment for delivering to me the item g_1 within 3 days" is represented by $C_2^0(me, you, 1, +\Delta^{p}_{O}C_1^0(z, me, 1, g_1 : 3))$, which may help "me" in the negotiation process with "you".

Definition 14. *In a Commitment Existence Notification the debtor x informs the creditor y about the existence of a specific commitment according to its defeasible theory D.*

$$C_2^0(x, y, 1, \star C_1^0(z, w, 1, p))$$

"If you promise me to keep it secret I will tell you if z is committed to me or not to deliver g_1" will be $C_2^2(me, you, C_1^0(you, me, 1, secret), \star C_1^0(z, me, 1, g_1))$, an example of a *confidentiality agreement.* This situation may arise during negotiations for a larger contract, when agents may need to divulge information about their operations to each other, also known as non-disclosure agreement.

Definition 15. *In a Conditional Notification the debtor x informs the creditor y about the existence of a specific commitment if condition q holds until t_i.*

$$C_2^0(x, y, q : t_i, ?C_1^0(z, w, 1, p))$$

Asking for represents a composition between a request commitment and an informing commitment, e.g., "Please tell me if the payment was made", represented as $C_2^1(me, bank, C_1^0(bank, me, 1, ?pay), 1)$.

Definition 16. *In an Asking For the debtor x asks the creditor y about the existence of a specific fact p.*

$$C_2^1(x, y, C_1^0(y, x, 1, ?p), 1)$$

4 Commitment life cycle

During its life cycle, a commitment may be in one of the following states: *open offer, active, released, breached, fulfilled, canceled, or failed* (figure 5), which are also useful to be considered from a legal perspective.

First consider a gratuitous promise $C_1^0(x, y, 1, p : t_{maturity}) : t_{expiration}$. Under the donative-promise principle, a simple, unrelied-upon gratuitous commitment is unenforceable since there is no consideration [8] or no element of exchange. Therefore, the breach of a C_1^0 commitment attracts only social sanctions or trust sanctions. The use of normative foundation of trust attached to a C_1^0 commitment serves to promote business relations. In case the creditor y has relied on the commitment, one can make use of the doctrine of *promissory estoppel.* This doctrine comes from the equity part of the law and it prevents one party from withdrawing a promise made to a creditor, if that creditor has relied on

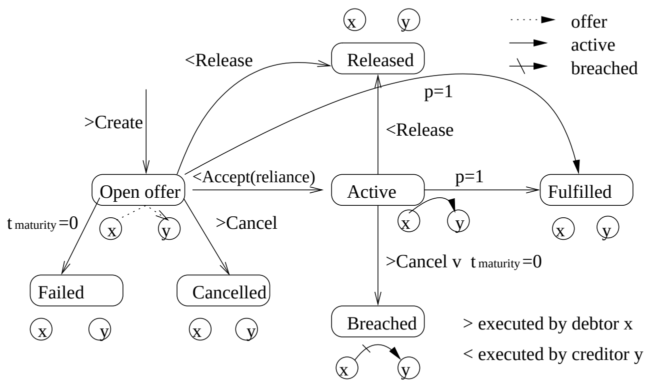


Fig. 5. The life cycle of a gratuitous commitment

that promise and acted upon it. The only remedy of contract law that can be applied in this case is *reliance damages* [8]. Also, the law stipulates that this reliance must be foreseeable. In the context of open agent systems we define a foreseeable fact as one which has been notified to the potential breacher. For instance, in a supply chain scenario, the creditor must notify the promiser that, based on the C_1^0 commitment, it has signed other contracts: "I inform you that, based on your gratuitous promise, I commit to deliver item g_1 to my client z within 3 days". This is represented by $C_2^0(me, you, 1, +\Delta_K^p C_1^0(me, z, 1, g_1 : 3))$. On the other hand, the estoppel is "a shield, not a sword". It cannot be used as the basis of an action of its own. Hence, we implement estoppel with defeaters.

$$\begin{aligned}
&\to_I^t promise(p : t_m, y) : t_i \\
&\Rightarrow_I^p riskProne : t_i \\
&\to_K^p promissoryEstoppel : t_i \\
r_0 &: promise(p : t_m, y) : t_i, riskProne : t_i \Rightarrow_A^t create(x, c) : t_i \\
r_1 &: create(x, c) : t_i \to_K^p c : t_i \\
r_2 &: c : t_i, t_m = t_i \to_K^p \neg c : t_i \\
r_3 &: c : t_i, cancel(x, c) : t_i \Rightarrow_K^p \neg c : t_i \\
r_4 &: c : t_i, release(y, c) : t_i \to_O^p \neg c : t_i \\
r_5 &: breached : t_i \Rightarrow_O^p relianceDamages : t_{i+3}, \neg c : t_i \\
r_6 &: specificPerformance : t_i \leadsto_O^p \neg c : t_i \\
r_7 &: execute(p) : t_i \Rightarrow_K^p p : t_{i+2} \\
r_8 &: assign(y, z, c) : t_i, c : t_i \Rightarrow_O^p \neg c : t_i, C_1^0(x, z, 1, p : t_m) : t_i \\
r_9 &: delegate(x, z, c) : t_i, c : t_i \Rightarrow_O^p \neg c : t_i, C_1^0(z, y, 1, p : t_m) : t_i
\end{aligned}$$

Fig. 6. Sample of rules for commitment operations

Possible operations on commitments: create, cancel, release, assign, and delegate (figure 6) are discussed next, considering their effect on a gratuitous commitment $c = C_1^0(x, y, 0, p : t_m)$. Similar rules are defined for other types of commitments, the main difference results from what acceptance means for each type of commitment. For instance, the acceptance of a gratuitous commitment means reliance and acted upon it, the aceptance of a unilateral contract means the execution of the required task, the acceptance of a bilateral contract means the creation of the required promise, etc.

Create. Consider that agent x has the intention to satisfy sentence p for agent y, until deadline t_m. Its policy is risk prone, meaning that it creates the gratuitous commitment c, while it has no guarantee that its partner will give something in exchange. Moreover, the interaction is made under the doctrine of promissory estoppel. The above intentions drive the agent to create the commitment c (rule r_0, which being transient, the *create* action is executed once). The creation of a commitment, an action typically undertaken by the debtor, is equivalent to an *open offer* in contract law. Therefore, it is derived only as persistence knowledge (rule r_1) and is not considered an obligation in this state[4].

Cancel. The debtor x may *cancel* a commitment with no penalties only if the commitment is an open offer (rule r_3). The *breached* state is reached when the time for accomplishing the promise elapses. This state activates the mechanism for computing reliance damages, which usually suppose the creation of another commitment or contrary-to-duty obligation[5]. In some situations, a commitment may be active even after it is breached, allowed by defining rule r_5 as defeasible. Therefore, a normative agent may block the derivation of that conclusion in order to force the execution of the specific commitment c (rule r_6)[6]. When the timeout of an open offer commitment expires, the state of the commitment becomes *failed* (rule r_2).

Release. If the acceptance has been made, this operation releases the debtor from its gratuitous commitment (rule r_4). The agent x executes p, but the effect is expected to be seen after two time steps (rule r_7). The defeasible rule r_7 leaves space to treat some exceptions.

Assign. The *assign* operation, transferring the rights held by the creditor y to another party, the assignee z, may be executed only by the creditor y and the state of the commitment is preserved (rule r_8). Common law favors the freedom of assignment, unless there is an express prohibition against it, requiring that it must occur in the present, to assign in the future having no legal effect.

Delegate. The *delegate* operation, transferring the duties held by the debtor x to another party z, is executed only by the debtor x and the state of the

[4] Equivalent to a proposed or attempted commitment.

[5] The sooner it notifies by executing the *cancel* operation, the lower the reliance damages.

[6] In common law [8] expectation damages and no specific performance are granted as the usual remedy in case of breach. Since contracts, more often, are essentially about profit, the granting of expectation damages provides an acceptable substitute to the innocent party. While the state of granting of a performance remedy would amount to doing unnecessary harm to the party who has committed the breach [6].

commitment is preserved (rule r_9). The creditor must be informed of the act of delegation. In case z breaches, the creditor y may elect to treat this failure as a breach of the original commitment and to sue the delegator x or to choose the role of a third party beneficiary.

In the case of the life cycle of a unilateral contract, the debtor x can revoke his commitment anytime before acceptance. When the condition q becomes true, the commitment becomes active. Until then, the debtor may cancel without considering this as a breach. Most courts now hold that creditor y must give notice of its acceptance after it has done the requested act. If it does not do that, the commitment that was formed by the act may be canceled without breach (of course, the debtor must return the money). Therefore, the acceptance of a C_1^1 commitment can be viewed as a compound operation: execution of the q and a fact notification $C_1^0(x, y, 1, +\partial_K^p q)$. Due to the late activation of the C_1^1 commitment the promiser x has maximal protection. What happens if the creditor executes a part of the q condition and notifies about this? The common law stipulates that an *option contract* was formed, which protects the creditor y from the debtor's ability to cancel the commitment (i.e. *partPerformance* $\rightarrow$ *optionContract, optionContract* $\leadsto \neg C_1^1(x, y, q, p : t_{maturity})$). If acceptance is late ($t_{issue} < t_{acceptance} < t_{maturity}$), it becomes a counter-offer and it creates the power of acceptance for the initial debtor x.

5 Using higher-order commitments

5.1 English auction

We illustrate the usage of commitments in the *English auction* (figure 7). According to contract law, when an item is put up for auction, this is usually not an offer, but rather a solicitation of offers (bids) or an invitation to treat. The English auction protocol uses the pattern "request a unilateral contract"[7]. Therefore, the auctioneer a has to compose a request commitment with a unilateral contract (f_1 in figure 8, where "-" is used to express existential quantification)

$$
\begin{aligned}
&r_{21} : deliver(g_1) : t_3 \rightarrow_K^p C_1^0(b, a, 1, +\partial_K^p g_1 : t_7)) : t_3\\
&r_{22} : deliver(g_1) : t_3 \Rightarrow_K^p g_1 : t_7\\
&r_{23} : g_1 : t_3 \rightarrow_O^p C_1^0(b, a, 1, +\partial_K^p g_1 : t_7)) : t_3\\
&r_{24} : pay : t_9 \rightarrow_A^t release(a, b, C_1^0(b, a, 1, 12 : t_9)) : t_9
\end{aligned}
$$

Fig. 7. Sample rules for English auction

[7] For the simplified Net bill protocol [9] which ignores the cryptography-related aspect and also the existence of a third party agent, unlike the complete version of the Net bill protocol [10] we would use the "request a conditional bilateral contract" $C_{1.5}^3(x, y, C_{1.5}^2(y, x, C_1^1(x, y, Deliver, EPO), receipt), 1)$.

$f_1: \quad C^2_{0.5}(a, -, C^1_1(-, a, g_1 : t_7, bid > 10 : t_9) : 3, 1) : t_1$
$f_2: \quad C^1_1(b, a, g_1 : t_7, 12 : t_9) : t_2$
$f_3: \quad C^1_1(b', a, g_1 : t_7, 11 : t_9)) : t_2$
$f_4: \quad C^{0.5}_2(a, b', 1, +\Delta^p_K \neg C^1_1(b', a, g_1 : t_7, 11 : t_9))) : t_3$
$f_5: \quad C^{0.5}_2(a, b, 1, +\Delta^p_O C^1_1(b, a, g_1 : t_7, 12 : t_9))) : t_3$
$f_6: \quad deliver(g_1) : t_3$
$f_7: \quad C^0_1(a, b, 1, +\partial^p_K g_1 : t_7)) : t_3$
$f_8: \quad C^0_1(b, a, 1, +\Delta^p_K g_1 : t_7)) : t_7$
$f_9: \quad C^0_1(b, a, 1, +\partial^p_K 12 : t_9)) : t_7$

Fig. 8. A trace in English auction

for item g_1 with starting price 10\$, and bids expected for 3 time steps. In case of accepting the bids, a has to deliver the item g_1 in 7 time steps, while b has to pay for it in 9 time steps.

Suppose that two bids are received (f_2 and f_3) at t_2, both open offers. Hence, at this stage, both b and b' may cancel their C^1_1 commitments without breach, and a also may cancel its $C^2_{0.5}$ commitment, because the inner commitment is not active yet (according to current practice in law). The above commitments reach the *active* state and they become obligations only if a accepts them. The bidders have made offers according to the auctioneer request regarding the deadline for sending bids and $t_{maturity}$. In other encounters they might react with different terms, which would be considered a counter-offer and a more complex form of negotiation would arise.

At t_3, when the deadline for receiving bids expires, a clears the auction, considering the bids that conform to the request and accepting the winning one (lower level aspects of coordination are not shown). It may explicitly reject one bid (f_4) and accept the other one (f_5). In a unilateral contract the completion of the requested act is necessary to indicate acceptance. Most courts now hold that creditor y must also give notice of its acceptance after it has done the requested act. Therefore, the acceptance of a C^1_1 commitment can be viewed as a compound operation: execution and a commitment notification. Due to the late activation of C^1_1 the promiser has maximal protection. At this time, the existence of the requested commitment C^1_1 is verified and $C^2_{0.5}$ is discharged, leaving C^1_1.

The defeasible derivation rule r_{22} allows to treat some exceptions[8]. When the partner informs that the item has arrived (f_8), the strict rule r_{24} fires, C^1_1 becomes *active*, and when the item arrives after 4 time steps b_1 releases it. With the payment made, the auctioneer would release the debtor b from its commitment (rule r_{24}), otherwise the mechanism for treating exceptions should be activated according to a's policy.

[8] For instance, due to an accident the item has not arrived.

5.2 Considering risk in the supply chain

Consider the contract between two agents *me* and *you*, with agent *me* having to deliver the item, while agent *you* having to pay for it. There is more than

Risk	**Commitments**	**Meaning**
risk prone	$C_1^0(me, you, 1, deliver) \wedge$ $C_0^2(me, you, C_1^0(you, me, 1, pay), 1)$	I commit to deliver the item and I request you to commit to pay for it
moderate risk prone	$C_1^0(me, you, 1, deliver) \wedge$ $C_0^1(me, you, pay, 1)$	I commit to deliver the item and I request you to pay for it
risk neutral	$C_{1.5}^1(me, you, C_1^0(you, me, 1, pay),$ $deliver)$	I commit to deliver the item if you commit me to pay for it
moderate risk averse	$C_1^1(me, you, pay, deliver)$	I commit to deliver the item after you pay for it
risk averse	$C_2^1(me, you, pay, C_1^0(me, you, 1, deliver))$	I will commit to deliver the item if you pay me

Table 1. Risk attitudes between two agents

one possibility to represent this process, depending on the commitments signed between them, identified by five levels of risk attitudes (table 1). Now consider

Risk	**Commitments**	**Meaning**
risk averse	$C_{2.5}^1(me, you, C_1^0(sup, me, 1, deliver'),$ $C_1^0(me, you, 1, deliver)$	If my supplier commits to deliver my input item, I commit to deliver my output item
risk neutral	$C_{2.5}^{2.5}(me, you, C_{1.5}^1(sup, me,$ $C_1^0(me, sup, 1, pay'), deliver'),$ $C_1^0(me, you, 1, deliver))$	If my supplier commits to deliver my input item if I promise him to pay, I commit to deliver my output item
risk prone	$C_{1.5}^{2.5}(me, you, C_1^1(sup, me, pay, deliver),$ $C_1^0(me, you, 1, deliver)$	If my supplier commits to deliver my input item if I pay it, I commit to deliver my output item

Table 2. Risk attitudes considering a third party

the situation when agent *me* is conditioned by its supplier *sup*. In order to deliver its output item, it has to obtain first its input item (table 2) with other possible attitudes towards risk.

Assuming agent *me* has a risk prone strategy ($\Rightarrow_I^p riskProne : t_i$), it will create commitments $C_1^0(me, you, 1, deliver)$ and $C_0^2(me, you, C_1^0(you, me, 1, pay), 1)$. The acceptance of $C_1^0(me, you, 1, deliver)$ appears when agent *you* relies on it and it also notifies agent *me* about this reliance[9]. Once the acceptance occured, the commitment reaches the active state ($\Rightarrow_O^p C_1^0(me, you, 1, deliver)$) and thus it becomes an obligation for agent *me*. On the other side, agent *you* has no obligation at all, knowing only that its partner has requested to promise to pay

[9] Such a notification may look like this: "I (agent *me*), based on a gratuitous promise, commit to deliver item g_1 to my client z within 3 days", represented by $C_2^0(you, me, 1, +\Delta_K^p C_1^0(you, z, 1, g_1 : 3))$.

for the item[10]. In case of a risk neutral strategy, the acceptance occurs at the creation of the inner commitment ($\rightarrow_A^t$ $create(you, C_1^0(you, me, 1, pay))$). Thus, each agent has one obligation: $\rightarrow_O^p$ $C_1^0(me, you, 1, deliver)$ for agent *me* and $\rightarrow_O^p$ $C_1^0(you, me, 1, pay)$ for agent *you*. In case of a risk averse strategy the acceptance of the unilateral contract is done by the completion of the requested act, in this case the payment. Therefore, agent *me* has the obligation to deliver the item only after it had received the payment ($pay \rightarrow_O^p C_1^0(me, you, 1, deliver)$). In table 2 agent *me* has the obligation to deliver its output item only in case it has active contracts with its supplier regarding its input item. A similar risk averse strategy can be adopted on the other side of the flow within the supply chain. In this situation, the contracts with the suppliers become active only if demand exists for the items, a part of the market fluctuations being taken by the supplier instead of *me*.

6 Related work and conclusions

Ideas from legal reasoning have been applied to social commitments [1, 7], but without the use of the contract law, although the rich semantics of higher-order commitments [7] introduces concepts like: ought, pledge, taboo, convention, collective commitment, obligation, claim, privilege, power, and immunity.

The declarative contracts using RuleML [11] use a semantic part for contracts, while contracts have already been represented with defeasible logic and RuleML [12]. By introducing commitments, we offer a more flexible solution for contract monitoring and for agents reasoning on current actions.

Causal logic has been used [13] for protocol engineering, leading to a formal method for protocol design, and more realistic commitments can also be modelled in event calculus [14]. Our commitments are addressed in a more contractual style, with the deadlines attached to commitments offering a more realistic approach from a contractual point of view.

Commitments between a network of agents have also been analyzed [3], but without time constraints. Our higher-order commitments are closer to the leveled commitment contracts [6], with different attitudes towards risk.

Verdicchio and Collombetti [15] treat the semantics of communicative acts in terms of social commitments, instead of the classical approach, with a precommitment similar to our commitment having the *open offer* state derived from contract law. Our higher-order commitments have a similar semantics to the derivative communicative acts [15], but we also cover the completion of the requested act.

By introducing defeasible commitments in the execution of contracts, we obtain two main advantages. On the one hand, agents can reason with incomplete information, including confidential contractual clauses. On the other hand, this framework is suitable for exceptions and legal reasoning: (i) concerning resolution of a dispute, strategies are explainable; (ii) skeptical mechanism; (iii) allows

[10] In the case of a moderate risk prone strategy, agent *me* requests agent *you* to effectively pay for the item and not only to promise to pay.

preferences; (iv) linear complexity; (v) fine-grained mechanism to deal with exceptions in the same manner for expected or unexpected ones.

7 Acknowledgments

We are grateful to the anonymous referees for useful comments. Part of this work was supported by the grant 27702-990 from the National Research Council of the Romanian Ministry for Education and Research.

References

1. Pasquier, P., Flores, R.A., Chaib-draa, B.: Modelling flexible social commitments and their enforcement. In Gleizes, M.P., Omicini, A., Zambonelli, F., eds.: Engineering Societies in the Agents World. LNAI 3451, Springer-Verlag (2005) 139–151
2. Mallya, A.U., Singh, M.P.: Modeling exceptions via commitment protocols. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, Netherlands, ACM Press (2005) 122–129
3. Wan, F., Singh, M.: Formalizing and achieving multiparty agreements via commitments. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, Netherlands, ACM Press (2005) 770–777
4. Governatori, G., Rotolo, A., Sartor, G.: Temporalised normative positions in defeasible logic. In: 10th International Conference on Artificial Inteligence and Law, Bologna, Italy (2005)
5. Letia, I.A., Groza, A.: Running contracts with defeasible commitments. In Dapoigny, R., ed.: IEA/AIE, Annecy, France (2006) to appear.
6. Sandholm, T., Lesser, W.: Leveled commitment contracts and strategic breach. Games and Economic Behavior **35** (2001) 212–270
7. Singh, M.P.: An ontology for commitments in multiagents systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7** (1999) 97–113
8. Craswell, R.: Contract law: General theories. In Bouckaert, B., Geest, G.D., eds.: Encyclopedia of Law and Economics, Volume III. The Regulation of Contracts. Cheltenham (2000) 1–24
9. Winikoff, M., Liu, W., Harland, J.: Enhancing commitment machines. In: Declarative Agent Languages and Technologies. (2004) 198–220
10. Cox, B., Tygar, J., Sirbu, M.: Netbill security and transaction protocol. In: Proceedings of the First USENIX Workshop on Electronic Commerce, New York (1995)
11. Grosof, B.: Representing E-Commerce rules via situated courteous logic programs in RuleML. Electronic Commerce Research and Applications **3**(1) (2004) 2–20
12. Governatori, G.: Representing business contracts in RuleML. Journal of Cooperative Information Systems **14**(2-3) (2005)
13. Chopra, A.K., Singh, M.P.: Contextualizing commitment protocols. In: 5th International Joint Conference on Autonomous Agents and Multiagent Systems, Hakodate, Japan, ACM Press (2006)
14. Yolum, P., Singh, M.P.: Reasoning about commitments in the event calculus: An approach for specifying and executing protocols. Annals of Mathematics and Artificial Intelligence **42**(1-3) (2004)
15. Verdicchio, M., Colombetti, M.: A commitment-based communicative act library. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, Netherlands, ACM Press (2005) 755–761

Using Dynamic Logic Programming to Obtain Agents with Declarative Goals – preliminary report

Vivek Nigam[⋆] and João Leite

CENTRIA, New University of Lisbon, Portugal
`vivek.nigam@gmail.com` and `jleite@di.fct.unl.pt`

Abstract. Goals are used to define the behavior of (pro-active) agents. It is our view that the goals of an agent can be seen as a knowledge base of the situations that it wants to achieve. It is therefore in a natural way that we use Dynamic Logic Programming (DLP), an extension of Answer-Set Programming that allows for the representation of knowledge that changes with time, to represent the goals of the agent and their evolution, in a simple, declarative, fashion. In this paper, we represent agent's goals as a DLP, discuss and show how to represent some situations where the agent should adopt or drop goals, and investigate some properties that are obtained by using such representation.

1 Introduction

It is widely accepted that *intelligent agents* must have some form of *pro-active* behavior [19]. This means that an intelligent agent will try to pursue some set of states, represented by its *goals*. At the same time, goals will serve as explanations for agent's *actions*. Goals have two distinct, though related, aspects: a *procedural* that can be identified with the sequence of actions that the agent attempts to perform in order to achieve a goal; and a *declarative* that can be associated with the set of states that the agent wants to bring about. In this paper we will focus on the declarative aspect of goals.

Recently, there has been an increasing amount of research devoted to the issue of declarative goals and their properties [18, 6, 13, 16, 17, 15]. Agent programming languages with declarative goals open up a number of interesting possibilities to the programmer, such as checking if a goal has been achieved, if a goal is impossible, if a goal should be dropped, i.e., if the agent should stop pursuing a goal, or if there is interference between goals [18, 15]. Having declarative goals also facilitates the task of constructing agents that are able to *communicate* them with other agents [13]. In [18, 15, 13] the reader can find examples illustrating the need for a declarative aspect to goals.

As dynamic entities, agents often must adopt new goals, and drop existing ones, and these changes in the adopted goals can be made dependent on the

[⋆] Supported by the Alβan Program, the European Union Programme of High Level Scholarships for Latin America, no. E04M040321BR

state of affairs. There has been a great deal of research to identify when an agent should change its goals [5, 16, 15, 18]. For example, an agent should drop a goal when it believes that the goal is no longer achievable (maybe represented by a *failure condition* [18]). As for adopting new goals, after a negotiation is successfully closed there may be new *obligations* [7] that the agents involved have committed to, that lead to the revision of the agent's goals and, possibly, the adoption of new ones.

In this paper, we will address the problem of representing and reasoning about dynamic declarative goals using a logic programming based approach.

In [12, 8], the paradigm of *Dynamic Logic Programming (DLP)* was introduced. According to *DLP*, knowledge is given by a series of theories, encoded as generalized logic programs[1], each representing distinct states of the world. Different states, sequentially ordered, can represent different time periods, thus allowing DLP to represent knowledge that undergoes successive updates. Since individual theories may comprise mutually contradictory as well as overlapping information, the role of *DLP* is to employ the mutual relationships among different states to determine the declarative semantics for the combined theory comprised of all individual theories at each state. Intuitively, one can add, at the end of the sequence, newer rules (arising from new or reacquired knowledge) leaving to *DLP* the task of ensuring that these rules are in force, and that previous ones are valid (by inertia) only so far as possible, i.e. that they are kept for as long as they are not in conflict with newly added ones, these always prevailing.

It is our perspective that the declarative goals of an agent can be seen as a knowledge base encoding the situations it wants to achieve. There has been, in the past years, an intense study of the properties of DLP to represent knowledge bases that evolve with time [2, 8, 11]. However, up to now, there hasn't been much investigation of how DLP could be used to represent, in a declarative manner, the goals of an agent. Since DLP allows for the specification of knowledge bases that undergo change, and enjoys the expressiveness provided by both strong and default negations, by dint of its foundation in answer-set programming, it seems a natural candidate to be used to represent and to reason about the declarative goals of an agent, and the way they change with time.

In this paper we will represent the *goal base* of an agent as a *Dynamic Logic Program*, and investigate some of its properties. Namely, we will see that the semantics of DLP will allow us to straightforwardly *drop* and *adopt* new goals by updating the goal base of the agent, and will allow those operations to be conditional on the current state of affairs.

Furthermore, an agent can distinguish between *maintenance and achievement* goals. A maintenance goal represents a state of affairs that the agent wants to hold in all states. For example, a person doesn't want to get hurt. An achievement goal represents a state of affairs that, once *achieved*, is no longer pursued. For example, an agent that has as goal to write a paper for a congress, after it believes it has written the paper, it should no longer consider this as a goal. Therefore,

[1] Logic programs with default and strong negation both in the body and head of rules.

to correctly define the conditions for dropping a goal, we also investigate how to express maintenance and achievement goals using DLP.

For our purpose, we will use a simple agent framework to be able to clearly demonstrate the properties obtained by using DLP. Agents in this framework are composed of data structures representing its beliefs, goals, committed goals (intentions) and reasoning rules. We propose three types of reasoning rules: **1)** Intention Adoption Rule: used to *commit* to a goal by adopting a plan to achieve it; **2)** Goal Update Rule: used to *update* an agent's goals using the DLP semantics; **3)** Intention Dropping Rule: used to *drop* previously committed goals.

The remainder of the paper is structured as follows: in the next Section we are going to present some preliminaries, introducing DLP and the agent framework we are going to use. Later, in Section 3, we are going to define the semantics of the goal queries and in Section 4 the reasoning rules. In Section 5, we discuss some situations related to when to drop and adopt new goals, and how to use the DLP semantics to represent these situations. In Section 6 we give a *simple* example of a multi-agent system illustrating how DLP could be used to represent goals, to finally draw some conclusions and propose some further research topics in Section 7.

2 Preliminaries

In this section we are going to give some preliminary definitions that will be used throughout the paper. We start by introducing the language and semantics of *Dynamic Logic Programming* and, afterwards, we introduce the simple *agent framework* that we will adopt to demonstrate our investigations.

2.1 Dynamic Logic Programming

Let $\mathcal{K}$ be a set of propositional atoms. An *objective literal* is either an atom A or a strongly negated atom $\neg A$. A *default literal* is an objective literal preceded by *not*. A *literal* is either an objective literal or a default literal. The set of objective literals is denoted by $\mathcal{L}_{\mathcal{K}}^{\neg}$ and the set of literals by $\mathcal{L}_{\mathcal{K}}^{\neg,not}$. A *rule* r is an ordered pair $Head\,(r) \leftarrow Body\,(r)$ where $Head\,(r)$ (dubbed the head of the rule) is a literal and $Body\,(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $Head\,(r) = L_0$ and $Body\,(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A *generalized logic program* (*GLP*) P, in $\mathcal{K}$, is a finite or infinite set of rules. If $Head(r) = A$ (resp. $Head(r) = not\,A$) then $not\,Head(r) = not\,A$ (resp. $not\,Head(r) = A$). If $Head\,(r) = \neg A$, then $\neg Head\,(r) = A$. By the *expanded generalized logic program* corresponding to the GLP P, denoted by $\mathbf{P}$, we mean the GLP obtained by augmenting P with a rule of the form $not\,\neg Head\,(r) \leftarrow Body\,(r)$ for every rule, in P, of the form $Head\,(r) \leftarrow Body\,(r)$, where $Head\,(r)$ is an objective literal[2]. Two rules r and r' are conflicting, denoted by $r \bowtie r'$, iff $Head(r) = not\,Head(r')$. An *interpretation*

[2] Expanded programs are defined to appropriately deal with strong negation in updates. For more on this issue, the reader is invited to read [9, 8]. From now on, and

M of $\mathcal{K}$ is a set of objective literals that is consistent i.e., M does not contain both A and $\neg A$. We define the set $\mathcal{I}$ as the set of all interpretations. An objective literal L is true in M, denoted by $M \vDash L$, iff $L \in M$, and false otherwise. A default literal $not\, L$ is true in M, denoted by $M \vDash not\, L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \vDash B$, iff each literal in B is true in M. Only an inconsistent set of objective literals, In, will entail the special symbol $\perp$ (denoted by $In \models \perp$). $\perp$ can be seen semantically equivalent to the formula $A \wedge \neg A$. An interpretation M of $\mathcal{K}$ is an *answer set* of a GLP P iff $M' = least\,(\mathbf{P} \cup \{not\, A \mid A \notin M\})$, where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an objective literal, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\, A$ by a new atom not_A. For notational convenience, we will no longer explicitly state the alphabet $\mathcal{K}$. As usual, we will consider all the variables appearing in the programs as a shorthand for the set of all their possible ground instantiations.

A *dynamic logic program* (*DLP*) is a sequence of generalized logic programs. Let $\mathcal{P} = (P_1, ..., P_s)$, $\mathcal{P}' = (P'_1, ..., P'_n)$ and $\mathcal{P}'' = (P''_1, ..., P''_s)$ be DLPs. We use $\rho\,(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $\mathbf{P}_1, ..., \mathbf{P}_s$, and $(\mathcal{P}, \mathcal{P}')$ to denote $(P_1, ..., P_s, P'_1, ..., P'_n)$ and $\mathcal{P} \cup \mathcal{P}''$ to denote $(P_1 \cup P''_1, ..., P_s \cup P''_s)$.

In the past years there have appeared several semantics for a DLP. We are going to use the *Refined Dynamic Stable Model* semantics defined below, because of its nice properties, as investigated in [9].

Definition 1 (Semantics of DLP). *[8, 1] Let $\mathcal{P} = (P_1, \ldots, P_s)$ be a dynamic logic program over language $\mathcal{K}$, A be an objective literal, $\rho\,(\mathcal{P})$, M' and $least(.)$ be as before. An interpretation M is a (refined dynamic) stable model of $\mathcal{P}$ iff*

$$M' = least\,([\rho\,(\mathcal{P}) - Rej(M, \mathcal{P})] \cup Def(M, \mathcal{P}))$$

Where:

$$Def(M, \mathcal{P}) = \{not\, A \mid \nexists r \in \rho(\mathcal{P}), Head(r) = A, M \vDash Body(r)\}$$
$$Rej(M, \mathcal{P}) = \{r \mid r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j \leq s, r \bowtie r', M \vDash Body(r')\}$$

It is important to notice that a DLP might have more than one stable model. Each stable model can be viewed as a *possible world* that follows from the knowledge represented by the DLP. We will denote by $SM(\mathcal{P})$ the set of stable models of the DLP $\mathcal{P}$. Further details and motivations concerning DLPs and its semantics can be found in [8].

2.2 Agent Framework

In this subsection we are going to define the agent framework[3] that we will use throughout this article. We will start by introducing the concept of *agent*

unless otherwise stated, we will always consider generalized logic programs to be in their expanded versions.

[3] The agent framework defined in this section could be seen as a modified (simplified) version of the agent framework used in the 3APL multi-agent system [4].

configuration, which consists of a *belief base* representing what the agent believes the world is, a *goal base* representing the states the agent wants to achieve, a set of *reasoning rules* and a set of *intentions* with associated *plans* representing the goals that the agent is currently *committed* to achieve. We are going to make precise, later in Section 4, how the reasoning rules of the agents are defined. We are considering that the agent has, at its disposal, a *plan library* represented by the set of plans, *Plan*. A plan can be viewed as a *sequence of actions* that can modify the agent's beliefs or/and the environment surrounding it, and is used by the agent to *try* to achieve a committed goal.

Our main focus in this paper is to investigate the properties of representing the goal base as a Dynamic Logic Program. We are not going to give the *deserved* attention to the belief base. We consider the belief base as a simple interpretation. However, a more complex belief base could be used. For example, we could represent the belief base also as a Dynamic Logic Program and have some mechanism such that the agent has an unique model for its beliefs[4]. Elsewhere, in [14], we explore the representation of 3APL agent's belief base as a DLP.

Definition 2 (Agent Configuration). *An agent configuration is a tuple $\langle \sigma, \gamma, \Pi, R \rangle$, where $\sigma \in \mathcal{I}$ is an interpretation representing the agent's belief base, γ a Dynamic Logic Program representing it's goal base, $\Pi \subseteq Plan \times \mathcal{L}^{\neg}$ the intentions of the agent and R the set of reasoning rules.*

We assume that the semantics of the agents is defined by a *transition system*. A transition system is composed of a set of *transition rules* that transforms one agent configuration into another agent configuration, in one computation step. It may be possible that one or more transition rules are applicable in a certain agent configuration. In this case, the agent must decide which one to apply. This decision can be made through a *deliberation cycle*, for example, through a priority among the rules. In this paper, we won't specify a deliberation cycle. An unsatisfied reader can consider a *non-deterministic* selection of the rules.

We are interested in knowing what an agent believes and what are its goals. To this purpose, we start by introducing, in the next definition, the *belief and goal query languages.*

Definition 3 (Belief and Goal Query Language). *Let $\phi \in \mathcal{L}^{\neg, not}$ and $\phi' \in \mathcal{L}^{\neg}$. The belief query language, $\mathcal{L}_{\mathcal{B}}$, with typical element β, and the goal query language, $\mathcal{L}_{\mathcal{G}}$, with typical element κ are defined as follows:*

$$\begin{array}{lll} \top \in \mathcal{L}_{\mathcal{B}} & B\phi \in \mathcal{L}_{\mathcal{B}} & \beta, \beta' \in \mathcal{L}_{\mathcal{B}} \text{ then } \beta \wedge \beta' \in \mathcal{L}_{\mathcal{B}} \\ \top \in \mathcal{L}_{\mathcal{G}} & G\phi' \in \mathcal{L}_{\mathcal{G}} & \end{array}$$

Notice that we don't include default literals in the goal query formulas. This is because we believe that it would only make sense for an agent to pursue a situation that the agent is completely sure when it is achieved. For example, if

[4] For example, a *belief model selector* that would select one of the stable models of the belief base to represent the agent's beliefs.

an agent had the goal of not (by default) failing an exam, $not\ fail$, it would be possible for the agent not to study for the exam and still satisfy this goal (considering that the agent is not a *genius*) by simply not checking its mark. On the other hand, default literals can be quite useful for the belief queries. For example, for cautious agents in emergency situations, if an agent is not sure that a place is safe ($not\ safe$), it could trigger the goal of moving to a safer location. We will explain better how this could be represented when we discuss goal adoption and dropping, in Section 5.

Now, we will start by defining the semantics of the belief query formulas ($\models_B$). The semantics of the goal query formulas ($\models_G$), one of the key interests of this paper, will be defined later in Section 3.

Definition 4 (Semantics of Belief Formulas). *Let $B\phi, \beta, \beta' \in \mathcal{L}_\mathcal{B}$ be belief query formulas and $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration. Then, the semantics of belief query formulas, $\models_B$, is defined as:*

$$\langle \sigma, \gamma, \Pi, R \rangle \models_B \top$$
$$\langle \sigma, \gamma, \Pi, R \rangle \models_B B\phi \Leftrightarrow \sigma \models \phi$$
$$\langle \sigma, \gamma, \Pi, R \rangle \models_B \beta \wedge \beta' \Leftrightarrow \langle \sigma, \gamma, \Pi, R \rangle \models_B \beta \text{ and } \langle \sigma, \gamma, \Pi, R \rangle \models_B \beta'$$

Although this is a quite simple agent framework it will be enough for the purpose of this paper.

3 Semantics of Agent Goal Bases

As defined in the previous section, we are considering the goal base of the agent as a Dynamic Logic Program. We will use the stable models of the goal base of the agent to determine the goals that it should achieve. Since the logic programs used in DLP use default negation, we can have situations where one DLP has more than one stable model, each internally consistent, but entailing contradictory conclusions between them. For example, consider a goal base consisting of the logic program with the following two rules:

$$a \leftarrow not\, \neg a. \qquad\qquad \neg a \leftarrow not\, a.$$

This program has two stable models, namely $\{a\}$ and $\{\neg a\}$. Even though each of them is consistent (recall that models are interpretations which, themselves, are consistent), they are contradictory in the sense that one entails a while the other entails $\neg a$. This contradiction could be seen as undesirable. However, as argued by Hindriks et al. in [6], the goal base of an agent doesn't have to be consistent since, for example, the goals of an agent can be achieved at different times. We add to this that these apparently contradictory goals can just be seen as alternative ones. The semantics of the intention adoption rules, defined below, makes sure that the agent doesn't concurrently pursue inconsistent intentions[5].

[5] [17] uses a default logic system to be able to express contradictory goals, but no mechanism to drop goals is proposed. We propose a system based on the stable models of the goal base, with the same expressiveness as the system in [17], and with the possibility of elegantly drop goals.

However, we shouldn't directly consider the stable models of the goal base (γ) of an agent as its goals, because the agent shouldn't consider a goal if it already believes that the goal is currently achieved. A naive way of solving this problem is to refine the stable models of the goal base by removing the goals that are entailed by the belief base: $GM = \{M \setminus \sigma \mid M \in SM(\gamma)\}$. But, by doing so, we *partially* lose expressiveness of having *conditional goals*. Consider the following illustrative example:

Example 1 (Conditional Goals). Let the goal base of an agent be the DLP composed of one GLP, with the intended meaning that the agent has as goal to buy a Ferrari if it won the prize, otherwise it would like to buy a Beetle. The goal of getting an insurance will depend on which car the agent will buy.

$$\begin{aligned} buy\_ferrari &\leftarrow win\_lottery. \\ buy\_beetle &\leftarrow not\, win\_lottery \\ get\_insurance &\leftarrow buy\_ferrari. \end{aligned}$$

We must consider the agent's belief base to determine what its goals are. Which car to buy will depend on whether it believes to have won or not the lottery, since obviously winning the lottery would not be a feasible goal for the agent.

The next definition formalizes an agent's *Goal Models*. The agent's Goal Models will be used to represent the agent's goals and they are obtained by refining the stable models of the agent's goal base in such a way that the agent takes in consideration its beliefs, and doesn't consider a formula as a goal if this formula is entailed by its belief base. In the previous example, if $win\_lottery$ is entailed by the beliefs of the agent, $buy\_ferrari$ would be one of its goals.

Definition 5 (Goal Models). *Let $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration. Then, the set of Goal Models (GM) of the agent is defined as:*

$$GM\,(\sigma, \gamma) = \{M \setminus \sigma \mid M \in SM((\gamma, \Psi(\sigma)))\}$$

where $\Psi(\sigma) = \{L \leftarrow \mid L \in \sigma\}$

Notice that, similarly to interpretations, the Goal Models are individually consistent, but two different goal models can be mutually contradictory. As argued previously, we want to express agents with contradictory goals. Therefore, to express the goals of an agent, we are going to use simultaneously all of its Goal Models.

The definition below formalizes the semantics of the goal query formulas.

Definition 6 (Semantics of Goal Query Formulas). *Let $G\phi, \kappa, \kappa' \in \mathcal{L}_{\mathcal{G}}$ be a goal query formula and $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration. Then, the semantics of goal query formulas, $\models_G$, is defined as:*

$$\begin{aligned} &\langle \sigma, \gamma, \Pi, R \rangle \models_G \top \\ &\langle \sigma, \gamma, \Pi, R \rangle \models_G G\phi \Leftrightarrow \exists M.(M \in GM(\sigma, \gamma) \wedge M \models \phi) \end{aligned}$$

The next proposition states that the agent cannot have a goal that is entailed by the belief base.

Proposition 1. *Let $\langle \sigma, \gamma, \Pi, R\rangle$ be an agent configuration, then:*

$$(\forall \phi \in \sigma).(\langle \sigma, \gamma, \Pi, R\rangle \not\models_G G\phi)$$

Proof. It is trivial from the way the Goal Models are constructed and by the Definition 6 of the Semantics of Goal Query Formulas

4 Reasoning Rules

We now define the types of reasoning rules an agent can have. We begin following [17], introducing the *Intention Adoption Rule* that is used by the agent to commit to a goal by associating a plan to it.

Definition 7 (Intention Adoption Rules). *Let $\beta \in \mathcal{L}_{\mathcal{B}}$ be a belief query formula and $\kappa \in \mathcal{L}_{\mathcal{G}}$ be a goal query formula, and $\pi \in Plan$ be a plan. The Intention Adoption Rules is defined as, $\kappa \leftarrow \beta \mid \pi$. We will call, β the guard of the rule and κ the head of the rule.*

Informally, the semantics of the Intention Adoption Rules is that if the goal base satisfies the head of the rule ($\kappa = G\phi$) and the agent beliefs in the guard (β) of the rule, the plan π is adopted to try to achieve the goal in the head of rule by adding the pair (π, ϕ) to the agent's intention base. However, as discussed by Bratman in [3], a *rational agent* shouldn't incorporate new intentions if it *conflicts* with the current intentions. For example, a rational agent wouldn't adopt the intention of going on vacations if it has committed to clean its house.

Taking this into account, we now formalize the semantics of the intention adoption rules.

Definition 8 (Semantics of Intention Adoption Rules). *Let $\langle \sigma, \gamma, \Pi, R\rangle$ be an agent configuration, $\kappa \leftarrow \beta \mid \pi \in R$, is an Intention Adoption Rule, where $\kappa = G\phi$, and $\Pi = \{(\pi_1, \phi_1), \ldots, (\pi_n, \phi_n)\}$.*

$$\frac{\langle \sigma, \gamma, \Pi, R\rangle \models_G \kappa \qquad \langle \sigma, \gamma, \Pi, R\rangle \models_B \beta \qquad \{\phi_1, \ldots, \phi_n, \phi\} \not\models \bot}{\langle \sigma, \gamma, \Pi, R\rangle \longrightarrow \langle \sigma, \gamma, \Pi \cup \{(\pi, \phi)\}, R\rangle}$$

Notice that the condition of consistency of the agent's intentions is maybe not yet the best option to avoid irrational actions, Winikoff et al. suggest, in [18], that it is necessary also to analyze the *plans* of the agent, as well as the *resources* available to achieve the intentions. However, this is out of the scope of this paper. The reader can also notice that the *conjunction of goals* cannot be expressed by only considering the intention adoption rule. It is necessary to increment the goal base of the agent. Consider that we want to program an agent with the following goal $a_1 \wedge, \ldots, \wedge a_n$. We can express this goal by having the following rules in the goal base $conj\_a_s \leftarrow a_1, \ldots, a_n$ and $conj\_a_s \leftarrow$, where $conj\_a_s$ is a new variable in the goal base. The goal $conj\_a_s$ will only be achieved if the conjunction $a_1 \wedge, \ldots, \wedge a_n$ is true.

We have just introduced a rule to adopt new intentions. Considering that intentions are committed goals, if the goal that the intention represents is no longer pursued by the agent, it would make sense to drop it. Therefore, we introduce into our agent framework the *Intention Dropping Rule*. Informally, the semantics of this rule is to remove from its intention base, any intention that is no longer supported by the goal base of an agent. The next definition formalizes this idea.

Definition 9 (Intention Dropping Rule). *Let $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration, where $\{(\pi, \phi)\} \subseteq \Pi$. Then:*

$$\frac{\langle \sigma, \gamma, \Pi, R \rangle \not\models_G G\phi}{\langle \sigma, \gamma, \Pi, R \rangle \longrightarrow \langle \sigma, \gamma, \Pi \setminus \{(\pi, \phi)\}, R \rangle}$$

As the intention dropping rule is defined, the agent could stop executing a plan if a goal is no longer entailed by the goal base. Stopping abruptly the execution of the plan could be undesired since there might be some *cleaning actions* to be taken after the goal is achieved. For example, if an agent's goal is to *bake a cake*, it would execute an appropriate plan, gathering the ingredients, the utensils, and setting up the oven. After the cake is baked the agent would still have to wash the utensils and throw the garbage away, these actions could be seen as clean up actions. To handle this issue, we could propose a more complex system of intentions, where there would be two plans associated with the committed goal, one used to achieve the goal and another used to do the *cleaning up*. When the goal is achieved the agent would execute the cleaning up plan. However, this issue is not our main interest here in this paper, and therefore we will limit our system to the intention dropping rule proposed in the definition above.

To be able to use the update semantics of DLP it is interesting to have a rule that can update the goal base of an agent with a generalized logic program. We will call this rule as *Goal Update Rule*. We will investigate in the Section 4, how to use the Goal Update Rule to adopt, drop or modify goals.

Definition 10 (Goal Update Rule). *Let P be a Generalized Logic Program and $\beta \in \mathcal{L}_\mathcal{B}$ be a query formula. The Goal Update Rule is defined as the tuple, $\langle \beta, P \rangle$. We will call β as the precondition of the goal update rule.*

Informally, the *semantics* of the goal update rule $\langle \beta, P \rangle$, is that when the precondition, β, is satisfied the goal base of an agent is updated by the generalized logic program P.

Definition 11 (Semantics of Goal Update Rules). *Let $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration, the semantics of a Goal Update Rule, $\langle \beta, P \rangle \in R$ is given by the transition rule:*

$$\frac{\langle \sigma, \gamma, \Pi, R \rangle \models_B \beta}{\langle \sigma, \gamma, \Pi, R \rangle \longrightarrow \langle \sigma, (\gamma, P), \Pi, R \rangle}$$

5 Adopting and Dropping Goals

In this section we are going to investigate how to represent, in our system, situations where an agent has to adopt or drop goals. We begin, in Subsection 4.1, by investigating how to represent failure conditions for goals. We will also define, in this Subsection, how to represent maintenance and achievement goals, since they are important concepts to be analyzed by an agent when it is intending to drop a goal. Later, in Subsection 4.2, we discuss some possible motivations of why an agent should adopt a goal and also investigate how to represent these motivations in our agent framework. Finally in Subsection 4.3, we identify some further properties of our framework.

5.1 Goal Dropping

In this subsection, we are going to investigate some situations where the agent must *drop a goal* and discuss how this could be done with our agent framework.

Winikoff et al. in [18], suggests some properties that the agent should have with respect to its goals, one of these properties is being able to define *failure conditions*. The idea is that when the failure condition is true the goal should be dropped and, furthermore, the agent should remove it from its intention base in case it had committed to it.

We can easily define failure conditions for goals using Dynamic Logic Programs, since failure conditions can be viewed as conditional goals. Consider the following example.

Example 2. Consider an agent that has to write a paper until a deadline of a conference. We could represent this situation using the following DLP, composed by a single GLP with a single rule, $write\_paper \leftarrow not\ deadline\_over$. The agent will consider $write\_paper$ as a goal only if the deadline is not over.

Another situation where the agent should drop a goal (or an intention) is when the goal (or intention) has been achieved, i.e., when the belief base entails the goal (or intention). By Proposition 1, we have that the agent will never entail a goal formula that is believed to be achieved. Hence, the agent can use the Intention Dropping Rule to drop intentions that are no longer goals of the agent.

Up to now we haven't explored the full expressiveness of Dynamic Logic Programs, by the simple fact that we didn't need, in any of the examples, the update semantics of DLP. We are going to use the semantics of DLP to be able to construct agents that can have *maintenance* as well as *achievement* goals.

In what circumstances an agent should drop a goal will depend in which type of goal it is. If it is an achievement goal, once it is achieved the goal must be dropped and not pursued in the future anymore. And if it is a maintenance goal, it will only be dropped when it is currently entailed by the agent's beliefs. But if in the future the goal is no longer entailed by its belief base, the agent will have to pursue this goal once more.

To be able to differentiate between these types of goals, we are going to define a special predicate, only appearing in the goal base, with signature, *maintenance(.)*, stating that the goal as argument is a maintenance goal. The following definition makes this precise.

Definition 12 (Maintenance and Achievement Goals). *Let $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration. We will call the goal ϕ as a maintenance goal iff*

$$\langle \sigma, \gamma, \Pi, R \rangle \models_G G maintenance\,(\phi) \wedge \langle \sigma, \gamma, \Pi, R \rangle \models_G G\phi$$

We call the goal ϕ an achievement goal iff

$$\langle \sigma, \gamma, \Pi, R \rangle \not\models_G G maintenance\,(\phi) \wedge \langle \sigma, \gamma, \Pi, R \rangle \models_G G\phi$$

We are going to use the semantics of DLP to define a *goal update operator* that updates the goals of the agent by dropping the achievement goals that have been achieved. The idea is to apply the goal update operator whenever the belief base of the agent is changed (this could be done by a deliberation cycle).

Definition 13 (Goal Update Operator - Ω). *Let $\langle \sigma, \gamma, \Pi, R \rangle \longrightarrow \langle \sigma', \gamma', \Pi', R \rangle$ be a transition in the transition system, where $\langle \sigma, \gamma, \Pi, R \rangle$ and $\langle \sigma', \gamma', \Pi', R \rangle$ are agent configurations, and $\Gamma(\sigma) = \{not\ L \leftarrow not\ maintenance(L) \mid L \in \sigma\}$. We define the goal update operator, Ω, as follows:*

$$\Omega(\gamma, \sigma') = \gamma' = (\gamma, \Gamma(\sigma'))$$

We must be sure that with the *goal update operator* defined above, new goals are not created and only the goals that have to be dropped are removed from the Goal Models. The next theorem states that when the goal update operator is used, no achievement goals that are achieved will be entailed by the agent, regardless of its future beliefs. For example, consider that an agent has achieved a goal ϕ and has updated its goal base with the goal update operator. If the agent doesn't adopt ϕ as a goal once more, or changes its status to a maintenance goal, ϕ will not be a goal of the agent even if in the future, the agent's belief base doesn't entail ϕ.

Theorem 1. *Let $\langle \sigma, \gamma, \Pi, R \rangle$ be an agent configuration and σ' be another belief base, such that $\langle \sigma, \gamma, \Pi, R \rangle \models_G G\phi$ and $\sigma' \models \phi$. Then:*

$$(\forall \sigma'' \in \mathcal{I}).(\langle \sigma'', \gamma', \Pi, R \rangle \not\models G maintenance(\phi) \Rightarrow \langle \sigma'', \gamma', \Pi, R \rangle \not\models G\phi)$$

where $\gamma' = \Omega(\gamma, \sigma')$.

Proof. Proof: Since $\sigma' \models \phi$ the goal update operator will update γ with a rule r, $\{not\ \phi \leftarrow not\ maintenance(\phi)\}$. As $\langle \sigma'', \gamma', \Pi, R \rangle \not\models G maintenance(\phi)$, the rule r will be activated rejecting all the rules with head, ϕ. Hence $\langle \sigma'', \gamma', \Pi, R \rangle \not\models G\phi$.

By proposition 1 we have that the maintenance goals will not be entailed by the agent if it believes that it is currently achieved.

5.2 Goal Adoption

Agents often have to adopt new goals. The reasons for adopting new goals can be varied, the simplest one, when dealing with pro-active agents, would be because the agent doesn't have any goals and it is in an *idle* state.

We follow [16], and distinguish two motivations behind the adoption of a goal: *internal* and *external*. Goals that derive from the desires of the agent, represented by *abstract goals*, have an internal motivation to be adopted. External motivations, such as *norms*, *impositions* from other agents, and *obligations*, can also be a reason for the agent to adopt new goals. An example of a norm, in the daily life, is that a person should obey the law. Obligations could derive from a *negotiation* where an agent commits to give a service to another agent e.g. your internet provider should (is obliged to) provide the internet connection at your home. Agents usually have a *social point of view* e.g. a son usually respects his father more than a stranger, and it may be the case that an agent imposes another agent some specific goals e.g. a father telling the son to study.

Dignum and Conte discuss, in [5], that an agent usually has *abstract goals* that are usually not possible to be achieved by a simple plan, but the agent believes that these abstract goals can be approximated by a set of *concrete goals*. Notice that the beliefs of the agent must be taken in consideration to adopt new concrete goals. For example, if an agent has the *desire* to obey the law and it believes that if it drives too fast it will break the law, it might have the goal of driving slower. On the other hand, it would be *acceptable* for the agent to talk on the mobile phone while driving a car, if an agent believes that by doing so it is not breaking the law, even though, by doing so, it might be *violating* the law.

Using DLP as the goal base of an agent we can *partially* simulate this behavior. Consider an agent with the a goal base consisting of one GLP, $\{\neg drive\_fast \leftarrow obey\_law; obey\_law \leftarrow; maintenance(obey\_law) \leftarrow\}$. As the agent will have the abstract maintenance goal of obeying the law (however there might be no plan to achieve it), it will try not to drive fast.

To be able to commit to obligations, changes in norms, or changes in desires, we need to be able to change the goal base during execution. For example, if a new deal is agreed to provide a service to another agent, the agent must entail this new obligation. By using the Goal Update Rule, an agent can update its goal base in such a way that it can *incorporate* new goals in several situations:

Adopt New Concrete Goals - As discussed previously, the agent may have some desires that can be represented by abstract goal κ that is usually not really achievable, but the agent believes that it can be approximated by some concrete goals $(\kappa_1, \ldots, \kappa_n)$. Consider that the agent learns that there is another concrete goal κ_l that, if achieved, can better approximate the abstract goal, κ. The agent can update its goal base using the following Goal Update Rule, $\langle concrete\_goal(\kappa_l, \kappa), \{\kappa_l \leftarrow \kappa\}\rangle$, as κ is a goal of the agent, it will activate the new rule, hence the new concrete goal, κ_l, will also be a goal of the agent;

Norm Changes - Consider that the agent belongs to a *society* with some norms that have to be obeyed $(norm_1, \ldots, norm_n)$ and furthermore that there is a change in the norms. Specifically, the $norm_i$ is changed to $norm'_i$, hence the agent's goal base must change. We do this change straightforwardly, using the goal update rule, $\langle change(norm_i, norm'_i), \{not\ norm_i \leftarrow; norm'_i \leftarrow\}\rangle$. This update will force all the rules, r, with $Head(r) = norm_i$ to be rejected and $norm_i$ will no longer be a goal of the agent. Notice that there must be some coherence with the change in the norms. For example, the agent shouldn't believe that on $change(norm_i, norm_j)$ and at the same time on $change(norm_j, norm_i)$;

New Obligations - Agents are usually immersed with other agents in an environment and, to achieve certain goals, it might be necessary to negotiate with them. After a negotiation round, it is normal for agents to have an agreement that stipulates some conditions and obligations (e.g. in *Service Level Agreements* [7]). The agent can again easily use the goal update rules to incorporate new obligations, $\langle obligation(\phi)$, $\{\phi \leftarrow\}\rangle$, as well as *dismiss* an obligation when an agreement is over, $\langle \neg obligation(\phi)$, $\{not\ \phi \leftarrow\}\rangle$;

Impositions - Agents not only negotiate, but sometimes have to *cooperate* with or *obey* other superior agents. This sense of superiority is quite subjective and can be, for example, the obedience of an employee to his boss, or a provider towards his client. It will depend on the beliefs of the agent to decide if it should adopt a new goal or not, but this can be modeled using the goal update rule, $\langle received(achieve, \phi, agent_i) \wedge obey(agent_i)$, $\{\ \phi \leftarrow\}\rangle$. Meaning that if it received a message from $agent_i$ to adopt a new goal ϕ, and the receiving agent believes it should obey $agent_i$, it will update its goal base. Notice that more complex hierarchy could be achieved by means of *preferences* between the agents. However, it would be necessary to elaborate a mechanism to solve possible *conflicts* (e.g by using Multi-Dimensional Dynamic Logic Programming [10]).

5.3 Further Properties

We still can identify some more properties that could be elegantly achieved by using the goal update rule:

Defining Maintenance and Achievement Goals We can define a goal as a maintenance goal if a certain condition is satisfied. For example, an initially single male agent finds the woman agent of its life and marries it. After this is achieved, it might like to be married with this agent until the end of its life. This can be represented by the goal update rule $\langle married(girl)$, $\{$ $married(girl) \leftarrow; maintenance(married(girl)) \leftarrow\}\rangle$. The opposite can also be easily achieved, using the goal update rule. A goal that initially was a maintenance goal can be dropped or switched to an achievement goal. For example, consider that the previous agent had a fight with its agent wife and, after the divorce, it doesn't want to marry again. This can be represented by the goal update rule, $\langle divorce(girl)$, $\{$ $not\ married(girl) \leftarrow$; $not\ maintenance(married(girl)) \leftarrow\}\rangle$. We define a new achievement or modify a maintenance goal to an achievement by using the following goal update rule $\langle achieve(\phi)$, $\{\ \phi \leftarrow$; $not\ maintenance(\phi) \leftarrow\}\rangle$;

Defining and Modifying Failure Conditions and Conditional Goals - As discussed, failure conditions are used to define when a goal has to be dropped. It is possible that the agent is not aware of all the failure conditions for a goal, or there has been a change in the environment such that the previous failure is not enough or, furthermore, it is not a valid failure condition anymore. Using the goal update rule, we are able to define *new*, *modify* or even *eliminate* failure conditions. Consider the example where the agent has to write a paper until a deadline and the deadline is postponed, we can use the following goal update rule, $\langle postponed\_deadline, \{write\_paper \leftarrow postponed\_deadline\}\rangle$. Conditional goals can be defined using a similar goal update rule.

6 Example

Consider a scenario containing two agents, a *father* and a *son*. Furthermore consider that the father agent is the head of a *mob family*. The son agent wants to obey the law but only if by doing so he doesn't disobey its father. Obeying the law can be viewed as an abstract goal that will be approximated by more concrete goals. These concrete goals can also been seen as the norms that the society imposes on the son agent. However, according to his social viewpoint, his father is more important than the society itself.

The goal base of the son agent can be represented by the following DLP:

$$\begin{aligned}
\neg kill &\leftarrow obey\_law, not\, disobey\_father.\\
disobey\_father &\leftarrow received\left(father, \phi, command\right), not\, \phi.\\
\phi &\leftarrow received\left(father, \phi, command\right).\\
obey\_law &\leftarrow .\\
maintenance\left(obey\_law\right) &\leftarrow .
\end{aligned}$$

Considering an initially empty belief set, the son agent has a unique Goal Model, namely $\{maintenance(obey\_law), obey\_law, \neg kill\}$. Consider that, subsequently, his father orders him to kill one of the mobsters of the rival family. Hence, the son receives the achievement goal of killing, modifying its beliefs to $\{received(father, kill, command)\}$. Therefore, the Goal Model of the son agent changes to $\{maintenance\ (obey\_law),\ obey\_law, kill\}$[6].

The son agent, after committing to the goal of killing, will create a plan to achieve it and, after executing the plan (killing the mobster), the agent updates its goal base with the rule

$$not\, kill \leftarrow not\, maintenance\left(kill\right).$$

And the Goal Model of the son is again $\{maintenance(obey\_law), obey\_law, \neg kill\}$. Consider now that the politicians, being annoyed by the gambling in city,

[6] Notice that $\neg kill$ is not in the Goal Model because we are using the expanded version of the GLPs

resolved to consider it illegal. Accordingly, the goal base of the son is updated with the GLP consisting of the following rule:

$$\neg gamble \leftarrow obey\_law, not\, disobey\_father.$$

The Goal Model of the son agent would change to $\{maintenance(obey\_law), obey\_law, \neg kill, \neg gamble\}$. However, his father, not being happy with this decision, orders his son to continue the gambling activities. Hence, $\{received(gamble, \phi, command)\}$ is added to his beliefs and the Goal Model of the son changes to

$$\{maintenance(obey\_law), obey\_law, \neg kill, gamble\}$$

This example illustrates how a programmer can use the Goal Update Rule to represent changes in the norms (considering gambling illegal) and use DLPs to represent concrete goals (not killing and not gambling). Furthermore, we could represent, in this small scenario, a social point of view of an agent (the son's social point of view) and how to give the *correct preference* on the goals according to this view.

7 Conclusions

In this paper, we introduced a simple agent framework with the purpose of introducing the agent's goal base as a Dynamic Logic Program. We investigated some properties of this framework. We were able to express, in a simple manner, conditional, maintenance and achievement goals, as well as identify some situations where the agent would need to adopt and drop goals, and how this could be done in this framework.

Since the objective of this paper was to investigate the use of DLP as the goal base of an agent, we didn't investigate what additional properties we could have by also using the belief base as a DLP. We also didn't give an adequate solution for conflicting intentions, since it would probably be also necessary to analyze the plans of the agent as well as its resources [18] to be able to conclude which goals to commit to.

Further investigation could also be done to solve possible conflicts in the social point of view of the agent. For example, if the agent considers the opinion of his mother and father equally, it would be necessary to have a mechanism to solve the conflicts since the agent doesn't prefer any one of them more than the other. [10] introduces the concept of *Multi Dimensional Dynamic Logic Programming* (MDLP) that could represent an agent's social point of view. Further investigation could be made in trying to incorporate the social point of view of an agent as a MDLP in our agent framework.

Even though this is still a preliminary report, we believe that DLP is a promising approach in which to represent the declarative goals of an agent, since it easily allows for the representation of the various aspects associated with agents' goals, and their updates, while enjoying a formal well defined semantics.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.
2. J. J. Alferes, J. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000.
3. M. Bratman. *Intentions, Plans and Practical Reason.* Harvard University Press, 1987.
4. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 2. Springer, 2005.
5. F. Dignum and R. Conte. Intentional agents and goal formation. In *Intelligent Agents IV*, volume 1365 of *LNAI*, pages 231–243, 1998.
6. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VII*, volume 1986 of *LNAI*, pages 228–243. Springer, 2000.
7. N. R. Jennings, T. J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Autonomous agents for business process management. *Applied Artificial Intelligence*, 14(2):145–189, 2000.
8. J. Leite. *Evolving Knowledge Bases.* IOS press, 2003.
9. J. Leite. On some differences between semantics of logic program updates. In *IBERAMIA'04*, volume 3315 of *LNAI*, pages 375–385. Springer, 2004.
10. J. Leite, J. J. Alferes, and L. M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. In *EPIA'01*, volume 2258 of *LNAI*, pages 276–289. Springer, 2001.
11. J. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In *Intelligent Agents VIII*, volume 2333 of *LNAI*. Springer, 2002.
12. J. Leite and L. M. Pereira. Generalizing updates: From models to programs. In *LPKR'97*, volume 1471 of *LNAI*, pages 224–246. Springer, 1998.
13. Á. F. Moreira, R. Vieira, and R. H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *DALT'03*, volume 2990 of *LNAI*, pages 135–154. Springer, 2004.
14. V. Nigam and J. Leite. Incorporating knowledge updates in 3apl. In *PROMAS'06*, 2006.
15. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & avoiding interference between goals in intelligent agents. In *IJCAI'03*, pages 721–726. Morgan Kaufmann, 2003.
16. B. van Riemsdijk, M. Dastani, F. Dignum, and J.-J. Ch. Meyer. Dynamics of declarative goals in agent programming. In *DALT'04*, volume 3476 of *LNAI*, pages 1–18, 2004.
17. M. B. van Riemsdijk, M. Dastani, and J.-J. Ch. Meyer. Semantics of declarative goals in agent programming. In *AAMAS'05.* ACM Press, 2005.
18. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *KR'02.* Morgan Kaufmann, 2002.
19. M. Wooldridge. *Multi-agent systems : an introduction.* Wiley, 2001.

A Collaborative Framework to realize Virtual Enterprises using 3APL

Gobinath Narayanasamy[1], Joe Cecil[2], and Tran Cao Son[1]

[1]Computer Science Department
New Mexico State University, USA
{gonaraya,tson}@cs.nmsu.edu
[2] Department of Industrial Engineering
New Mexico State University, USA
jcecil@nmsu.edu

Abstract. In this paper, we propose a collaborative framework to realize a Virtual Enterprise (VE) for the domain of Micro Assembly. The framework is developed using 3APL technologies [5] and employs the idea of viewing Web-Service composition as a planning problem [10]. We describe the implementation of the framework and experiment with two micro assembly work cells.

1 Introduction

In today's business world, being innovative and withstanding competitive pressure from contemporary business vendors are a key to success for any business vendors. With dynamic nature of consumer demands, business vendors often need a sophisticated mechanism to tap those momentous market demands. One such mechanism which will facilitate as well as satisfy the business vendors need is the concept of a Virtual Enterprise (VE). A VE is a conglomeration of different business vendors under one hood (to meet the market demands arising from consumers) by sharing their own resources and expertise, which – sometime – cannot be provided by a single business vendor. Each of the business vendors participating in a VE has different resource capabilities. Here a resource is anything that is necessary for the production of a product. It can be a machine, a software program, a component, a service, etc. Each resource might have a cost associated with it. Furthermore, there might be a resource, which can be used in the assembly of a product and is available in several places. The diversifying nature of a VE causes heterogeneity which slows down the process of forming collaborations among the vendors.

Our goal is to develop a framework that facilitates collaborations and seamless flow of information exchange among the partners in a VE. We explore this idea using the agent technologies provided by the 3APL framework [5].

We develop a prototype VE using the proposed collaborative framework in the Micro Assembly domain. Micro Assembly is the domain where parts in micron sizes are assembled using computer enabled micro assembly work cells. We target this domain for the following reasons: (*i*) it is considered as a better alternative to Micro

Electro Mechanical Systems (MEMS) where parts having varying material properties cannot be manufactured; (*ii*) it is a completely new methodology for developing products in manufacturing sector, and hence, not many business vendors possess the whole range of tools and resources to accomplish micro assembly related tasks such as micro assembly planning, simulation and actual physical implementation; and (*iii*) each business vendor possesses different micro assembly resources which – when used together – can accomplish most of the customer requests related to Micro Assembly.

As many parts in the Micro Assembly domain are assembled using computer programs and the VE is virtually available on the internet, we need a multi-agent development platform in which agents with various capabilities can be created. Each agent should have their own belief, capabilities, goals, and rules for reasoning. This platform should also facilitate the agent communication and collaboration. This is the reason why we choose 3APL as our implementation platform.

The paper is organized as follows: Section 2 provides a review of some past and recent developments of Virtual Enterprises using agent based approaches. Section 3 highlights the 3APL framework. Section 4 describes the collaborative system design. Section 5 discusses the development of collaborative framework using 3APL. Section 6 discusses VE formation for Micro Assembly domain using the proposed collaborative framework and Section 7 is the conclusions.

2 Literature Review

In this section, background information about virtual enterprises as well as a review of agent based systems is provided. Other issues such as agent communications, agent interaction protocols, and distributed problem solving approaches in agent based systems are also discussed.

In [12], it is observed that Co-operative or Concurrent Engineering (CE) techniques are the reason for forming collaborative working environment among company levels. In [3], a consortium of companies is called a Virtual Enterprise (VE) which allows for the development of a working environment to manage all or part of different resources towards achieving a common goal. Common information definition and sharing problem while forming Virtual Enterprises are discussed in [4]. The paper also discusses the issues of interaction among the companies that will agree upon a contract to form virtual enterprise.

In [8], the concept of forming Virtual Enterprises using agent based systems is proposed. In this conceptualization, partners of a virtual enterprise are considered as software agents. This paper also discusses different agent communication protocols such as KQML and KIF. A significant agent communication protocol proposed by US Defense Advanced Research Projects Agency's (DARPA) Knowledge-Sharing Effort known as Knowledge Query Management Language (KQML) is presented in [7]. The language includes variety of primitives, assertives, and directives which allow agents to query other agents, subscribe to other agents services, or find other agents for distributed problem solving. KQML assumes that each agent is built with its own knowledge bases. This allows other agents to extract information from the knowledge base of that particular agent. In the context of micro assembly, a sample KQML message between two agents, *Path_Planner* and *Service_Locater*, the former requests the latter for information about providers of a certain service with the help of ontological information is

(tell :sender *Path_Planner* :receiver *Service_Locater*
:ontology Micro_Assembly_ontology content publish(Service)).

In [6], Knowledge Interchange Format is emphasized. KIF is a language for interchanging knowledge between heterogeneous programs. KIF has a declarative semantics which allows agents to understand a KIF representation without any interpreters. It allows expressing arbitrary sentences using first order predicate calculus. It has constructs to represent knowledge in the domain, represent non monotonic reasoning rules and define objects, functions and relations. KIF has been employed in the development of the Process Specification Language (PSL), a language specifically designed to facilitate correct and complex exchange of process information among manufacturing systems [13].

In [10], it is observed that web services markup will allow agent technologies to efficiently capture the 'meta' data associated with the services and reason about them. This paves way for agent technologies to perform automated web services discovery, execution, composition and interoperation. In automated web services discovery, the software agent automatically discovers the web services based on user constraints, which is performed manually in the current World Wide Web (WWW). In automated web services execution, the software agent discovers the web services based on user constraints, understands the requirements for the services, and executes them automatically. In automated web services composition and interoperation, the software agent selects the required web services, compose and interoperate them to accomplish the requested complex task.

In [11], a need is identified to automate the process of discovering, executing, composing, and monitoring services. Automation refers to no human intervention and allows for the use of software agents. For a software agent to automatically process and execute a service, a machine understandable description of the service is required. One such language which provides descriptions that are machine understandable is OWL-S which is evolved as a collaborative work of BBN Technologies, Carnegie Mellon University, Nokia, Stanford University, SRI International, and Yale University.

In [2], the importance of using ontologies in manufacturing domain is explained. The paper emphasis on the need for developing richer ontological structures especially to the manufacturing domain so that more sophisticated intelligent applications can be developed.

3 3APL Language

An Abstract Agent Programming Language (3APL) developed at Universiteit Utrecht is a new agent oriented programming language for developing agents with cognitive capability, as given in [5]. The language comes with programming constructs that allows developing agents with complex mental states. A 3APL agent developed using this language is a tuple of the form <B, G, P, A> where,

- B is Belief base,
- G is Goal base,
- P is a set of Practical reasoning rules and
- A is an Action base.

Each of this component is briefly explained next.

3.1 Belief Base

A belief base encodes the agent knowledge about its operating environment and is a set of first order sentences. For example, a belief that a robot at room A is represented by the atom *at*(*Robot*, *RoomA*); other belief that a robot is not at the room *x* then it is at the room next to *x* is expressed by the sentence[1] $\forall x,y(\neg at(Robot,x) \wedge nextto(x,y) \Rightarrow at(Robot,y))$. Notice that a belief base can contain non-grounded sentences.

3.2 Goal Base

A goal base consists of *goals-to-do* goals. 3APL considers goals of procedural type. Under this view, a goal can be considered as an imperative program. A goal defines a plan of actions for an agent to execute. The language allows for the definition of simple and complex goals.

Simple goals (also called basic goals) are of three types: basic action, test goal, and achievement goal. For example, a simple goal like *inquireUDDI*() allows an agent to inquire the UDDI registry.

Complex goals (also called composite goals) are composed from basic goals and are used to specify complex actions such as sequences of actions, disjunctive goals, or non-deterministic choices, etc. Conventional programming constructs such as ';' and '+' are used to create complex goals. For example, "$goal_1$; $goal_2$" defines a sequence of goals and "$goal_1$+$goal_2$" defines a disjunctive goal.

3.3 Practical Reasoning Rules

A 3APL agent can manipulate its goals by using practical reasoning rules. These reasoning rules allow an agent to find plans, which help him/her achieve its goals. They also allow the agent to monitor its goal base. These rules facilitate the *practical reasoning* which an agent can use to decide (*i*) to adopt a plan for achieving a goal; (*ii*) to revise a plan if necessary. The set of practical rules is built from semi-goals and first order formulas where semi-goals are defined similar to goals using a new set of variables.

[1] The sentence might or might not be valid.

A practical reasoning rule is of the form

$$\pi \leftarrow \varphi \mid \pi',$$

where

- π is the head of a the rule,
- φ is the guard of the rule and
- π' is the body of the rule,
- Global variables are free first order variables in the head of the rule, and
- Local variables are non global first order variables in the body of a rule.

A practical rule $\pi \leftarrow \varphi \mid \pi'$ says that if the agent adopts some goal or plan π and believes that φ is true, then it may consider adopting π' as a new goal.

3.4 Action Base

Action base defines the set of primitive actions (or basic actions) that an agent can execute. This set of basic actions defines the capabilities of an agent with which an agent can change its mental state of belief about its working environment.

4 Framework Design

We follow the idea behind the design of this system follows the model proposed in [8] and [9]. We view each partner in a VE as an agent who has its own knowledge about the environment, its actions (basic and complex), its set of practical rules, and its own goals. A VE is a collection of agents who collaborate to achieve a common goal. As we have discussed above, most activities in the Micro Assembly domain are controlled by computer programs. As such, each partner is implemented as a software agent who can offer their services (or actions) to others. Our framework facilitates the communication between agents and allows users of the system to simulate the VE. The overall design of our framework is depicted in **Figure** 1.

Central to our system is a central manager agent which is a 3APL agent. This agent facilitates the communication between different agents and creating solutions for users' requests.

An agent can advertise its services in a service directory, which is implemented as a part of our system. A 3APL service directory agent provides other agents in the system the capability to find service provider(s) that can satisfy their needs. This agent communicates with other agents through the agent manager. In our implementation, each service is specified by its inputs and its execution method.

One issue in a collaborative framework is the semantically differences between different agents. This is also an issue in our framework. We follow others by addressing this issue using ontologies and develop ontologies for the Micro Assembly domain. To incorporate ontologies into our system, a 3APL agent is developed. This agent also communicates with other agents through the agent manager. We call this the meta-information of services.

We note that in [1], design and development of ontologies for physical devices are explained.

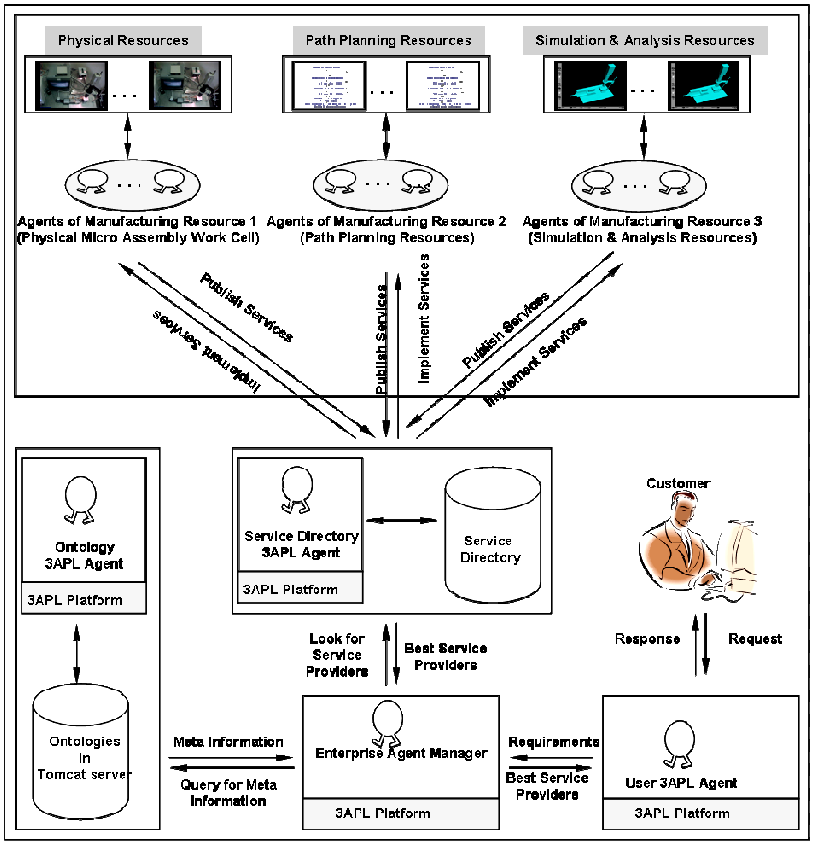


Figure 1. Collaborative System for Virtual Enterprise

5 Framework Implementation

This section discusses the implementation of the collaborative framework as shown in Figure 1. It consists of following agents:

1. User Agent
2. Virtual Enterprise Agent (or Enterprise Agent Manager)
3. Ontology Agent
4. Service Directory Agent and

5. Service Provider Agents

All these agents are implemented using 3APL and they run in 3APL platform. Plug-in programming construct is provided by 3APL platform so that agents can use the plug-in as their working environments and access the methods available in them. With the help of plug-ins, agents in 3APL platform can access the external JAVA methods, virtually allowing an agent to *execute* a service provided by another agent. For each agent in the our system, an associated plug-in is developed to assist the formation of Virtual Enterprise in real time. Detailed descriptions of 3APL agents used in the collaborative system are given below.

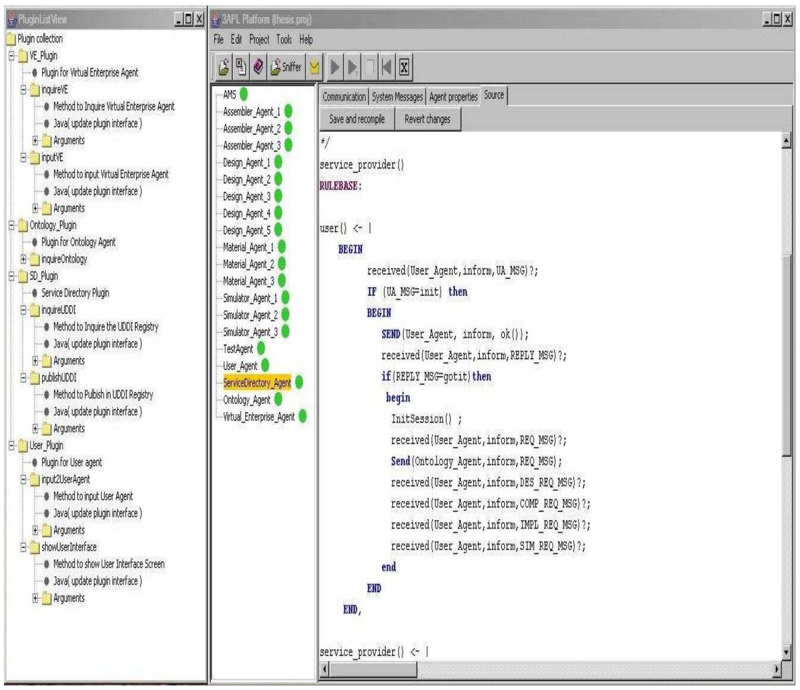
5.1 User Agent

User Agent provides the user interface to the collaborative system. This agent is probably the simplest agent in the system. It acts on behalf of real world entities such as human users, software applications, or even other business vendors who may need to accomplish a task.

5.2 Virtual Enterprise Agent

The Virtual Enterprise Agent coordinates the various activities in the collaborative framework. It is responsible for processing users' requests (from the user agent) and providing an initial solution (i.e. *plan*) for these requests. In the course of finding this solution, it queries the Ontology agent for meta-information and uses this information to find a list of best available service providers by querying the Service Directory agent.

The Virtual Enterprise Agent also serves as a search engine for other agents who need to find service providers for their own needs. **Figure** 2 shows a view of collaborative framework implemented in 3APL platform with developed plug-ins and participating software agents



```
*/
service_provider()
RULEBASE:


user() <- |
    BEGIN
          received(User_Agent,inform,UA_MSG)?;
          IF (UA_MSG=init) then
          BEGIN
            SEND(User_Agent, inform, ok());
            received(User_Agent,inform,REPLY_MSG)?;
            if(REPLY_MSG=gotit)then
             begin
              InitSession() ;
              received(User_Agent,inform,REQ_MSG)?;
              Send(Ontology_Agent,inform,REQ_MSG);
              received(User_Agent,inform,DES_REQ_MSG)?;
              received(User_Agent,inform,COMP_REQ_MSG)?;
              received(User_Agent,inform,IMPL_REQ_MSG)?;
              received(User_Agent,inform,SIM_REQ_MSG)?;
            end
          END
     END,


service_provider() <- |
```

Figure 2. Collaborative System for VE using 3APL

5.3 Ontology Agent

The Ontology Agent in the collaborative system provides the necessary meta-information for the VE agent to further process the input from the user agent. For demonstration purpose, some sample ontologies are created using Stanford’s Protégé editor. **Figure** 3 displays a part of the ontology developed for the Micro Assembly domain.

The ontologies developed for the collaborative system are deployed in a Tomcat web server. Any modifications to the existing ontologies are done through the ontology agent. This is achieved by means of a Ontology plug-in developed to assist the ontology agent. Ontology plug-in contains some basic functions for querying and modifying existing ontologies.

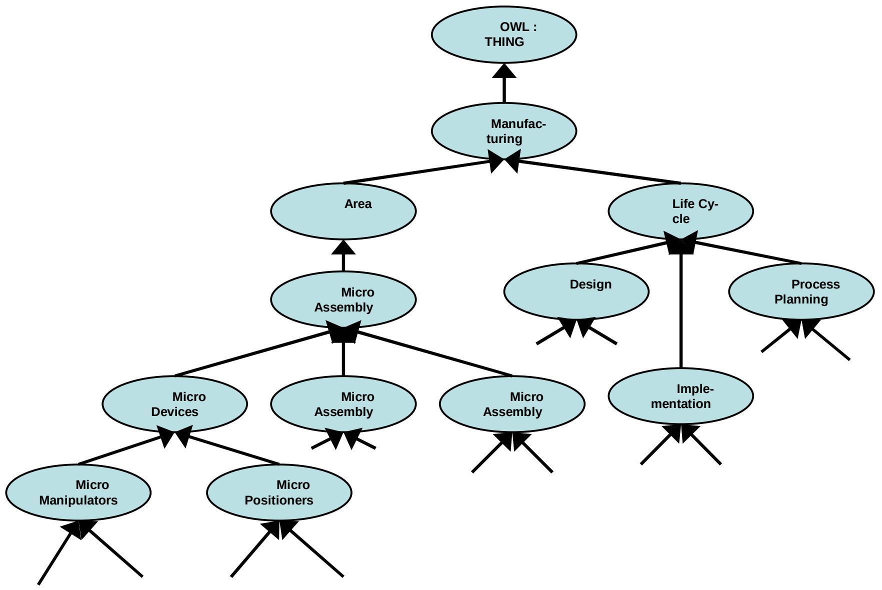


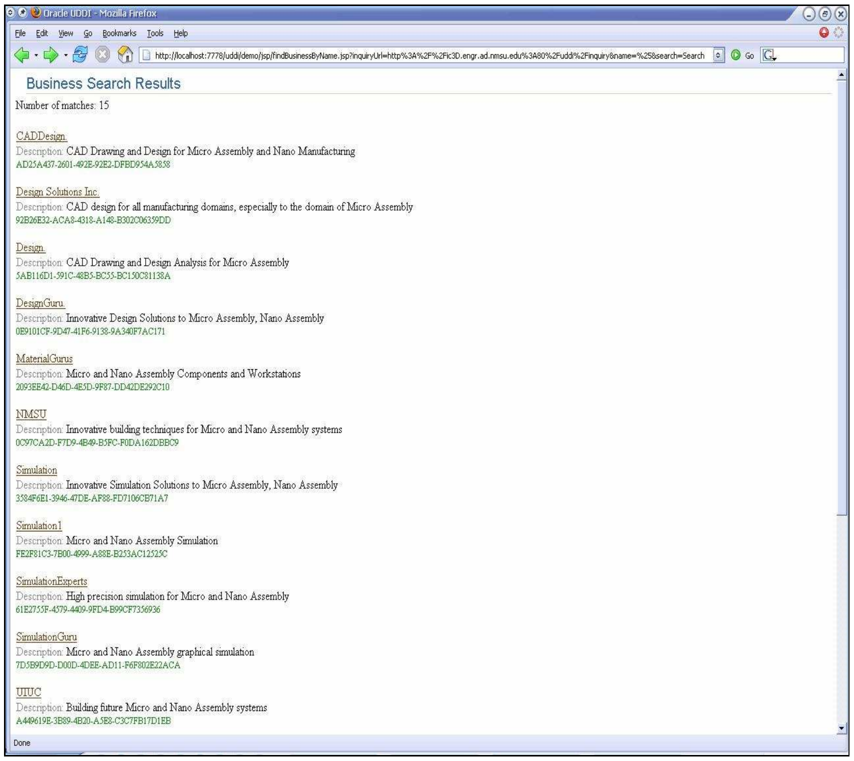
Figure 3. Sample Ontology

5.4 Service Directory Agent

The Service Directory Agent in the collaborative system is used to maintain a service directory where service provider agents will publish their services. This will facilitate other agents in the collaborative system, especially VE agents, to access the available services and use them to process the user agent's input. Oracle UDDI registry is used as the service directory in this collaborative system. Oracle UDDI registry comes along with the Oracle Application Server 10g. In this UDDI registry, instead of saving normal WSDL descriptions for services, OWL-S descriptions of services are saved. Requests from other agents for available services in the UDDI registry are made through this service directory agent. A service directory plug-in is developed for the agent to accomplish this task. The plug-in is developed with methods to connect to the service directory, publish OWL-S services in the service directory and inquire for available services. A screen shot of oracle UDDI registry is shown with some sample services is shown in **Figure** 4.

5.5 Service Provider Agent

Real business services in the collaborative system are provided by the service provider agents. Services provided by these agents range from software resources to actual physical implementation. Along with describing the service capabilities, the configurations of actual physical implementations are also described using OWL. A sample OWL description of a physical work cell can be accessed at http://128.123.245.156:9090/ontology/Device.owl


Oracle UDDI - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
Go
Done

http://localhost:7778/uddi/demo/jsp/findBusinessByName.jsp?inquiryUrl=http%3A%2F%2Fic3D.engr.ad.nmsu.edu%3A80%2Fuddi%2Finquiry&name=%25&search=Search

Business Search Results

Number of matches: 15

CADDesign.
Description: CAD Drawing and Design for Micro Assembly and Nano Manufacturing
AD25A437-2601-492E-92E2-DFBD954A5858

Design Solutions Inc.
Description: CAD design for all manufacturing domains, especially to the domain of Micro Assembly
92B26E32-ACA8-4318-A148-B302C06359DD

Design.
Description: CAD Drawing and Design Analysis for Micro Assembly
5AB116D1-591C-48B5-BC55-BC150C81138A

DesignGuru.
Description: Innovative Design Solutions to Micro Assembly, Nano Assembly
0E9101CF-9D47-41F6-9138-9A340F7AC171

MaterialGurus
Description: Micro and Nano Assembly Components and Workstations
2093EE42-D46D-4E5D-9F87-DD42DE292C10

NMSU
Description: Innovative building techniques for Micro and Nano Assembly systems
0C97CA2D-F7D9-4B49-B5FC-F0DA162DBBC9

Simulation
Description: Innovative Simulation Solutions to Micro Assembly, Nano Assembly
3584F6E1-3946-47DE-AF88-FD7106CB71A7

Simulation1
Description: Micro and Nano Assembly Simulation
FE2F81C3-7B00-4999-A88E-B253AC12525C

SimulationExperts
Description: High precision simulation for Micro and Nano Assembly
61E2755F-4579-4409-9FD4-B99CF7356936

SimulationGuru
Description: Micro and Nano Assembly graphical simulation
7D5B9D9D-D00D-4DEE-AD11-F6F802E22ACA

UIUC
Description: Building future Micro and Nano Assembly systems
A449619E-3B89-4B20-A5E8-C3C7FB17D1EB

Figure 4. Oracle UDDI registry showing sample services for the collaborative system

This allows the Virtual Enterprise agent to know more about the actual hardware implementation of devices. The collaborative system contains multiple service providers who will serve the needs of a user agent. Publication of services by these agents is accomplished through the service directory plugin, which provides methods for publishing the services into the UDDI registry.

6 Example Scenario

In this section, an example scenario is provided from the Micro Assembly domain to the collaborative system. Micro Assembly is considered as an alternative to MEMS based product development, where it is difficult to manufacture a product with different parts having varying properties. As explained in previous sections, it is completely a new area of product development where business vendors have limited number of sophisticated infrastructures and resources to accomplish a complete micro assembly based product development. In this application scenario, a user agent wants to assemble various micron sized parts (for eg. cams) on micron sized pins. Here, the goals of user agent are identification and formation of partnerships with potential business vendors and execution of their associated services.

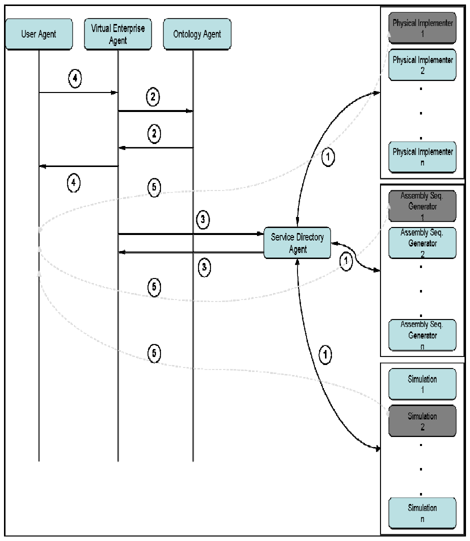


Figure 5. Interactions among the agents in the collaborative system

Possible interactions that will happen in this collaborative framework are listed below (refer to figure 5) and are elaborated subsequently.

1. Interactions between Service Directory Agent and Service Provider Agents.
2. Interactions between Virtual Enterprise Agent and Ontology Agent.
3. Interactions between Virtual Enterprise Agent and Service Directory Agent.
4. Interactions between User Agent and Virtual Enterprise Agent.
5. Interactions between Service Provider Agents and User Agent.

6.1 Service Directory Agent ←→ Service Provider Agents

To demonstrate this interaction, a set of service provider agents have been designed and implemented. These include service directory agents capable of providing

1. Services based on software applications such as assembly sequence generators, 3D path planners and virtual prototyping and analysis Environments
2. Services based on actual physical resources such as micro assembly work cells.

A brief description of some of these resources is provided along with their OWL and OWL-S descriptions.

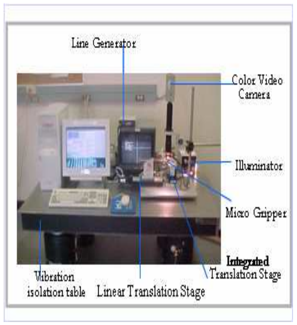
In order to assemble micron sized parts on micron sized pins, two micro assembly work cells as shown in figure 6, having different assembling capabilities are designed and developed. An ontology is developed to describe the capabilities in terms of work cell specifications. For example, work cell 1 is developed with gripper having the capability of assembling pins and cams in the size range of 100 – 200 microns (diameter) and a few millimeters in length. Due to the page limit, all OWL descriptions and grounding files necessary for the operation of the example are omitted. They are accessible from http://web.nmsu.edu/~gobinath/file.htm.

The maximum and minimum gripping force exerted by the gripper on its target object and its operating conditions are also described by an OWL element. The type of parts that the gripper can handle is given by the following OWL element

<parts_it_handle rdf:resource="#Cams"/>
<parts_it_handle rdf:resource="#Pins"/>

Similar to first micro assembly work cell, the second micro assembly work cell with tweezers is also described using OWL. This can be accessed at the URL http://128.123.245.156:9090/mawc2.owl. **Figure** 6 display two work cells used in our experiment.

~ ~



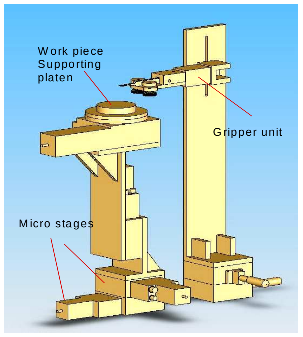


Figure 6. Micro Assembly Work Cells (Left: Work Cell 1, Right: Work Cell 2)

The assembly services of these two micro assembly work cells are made available as web services. As the assembly service requires physical components (cams and pins in this case) to be assembled, a software validation program is developed to validate the dimensions of input components with the capability of the respective micro assembly work cell. For example, in micro assembly work cell 1, the validation program validates the input by comparing the dimensions of the gripper and the parts to be assembled. If the validation program returns the positive results, further steps will be taken to ship the parts to the respective work cell location. This validation program is also made available as web services whose grounding information in OWL-S format is given in the above mentioned URL.

Apart from the work cells, virtual prototyping environments have been developed which form part of the VE resources. **Figure** 7 provides a snapshot of two virtual environments, which can be used to study alternates assembly and path plans, etc.

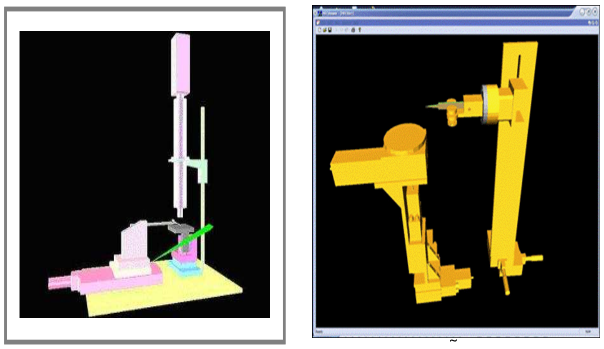


Figure 7. Different Virtual Environments (Virtual Environment 1, Virtual Environment 2)

These virtual environments are also accessible via web services. Service grounding information for one of the these VEs is described in OWL-S format and is available at http://web.nmsu.edu/~gobinath/file.htm.

Some of the software resources within the collaborative framework include micro assembly sequence generators as well as 3D path planners. Grounding information for one of the micro assembly sequence generators (determining an optimal sequence of assembling a target set of micro parts) using Genetic Algorithm is detailed below in 3APL format.

Sample message transfers that will take place during the interaction between a service provider agent (say, Micro Assembly Work Cell Provider) and the service directory (SD) agent while publishing a service are listed below:

Send(SD_Agent, inform, publish ()),
Send(SD_Agent, inform, serviceName (Micro Assembly Work Cell)),
Send(SD_Agent, inform,
    serviceDescription (http://128.123.245.156:9090/ontology/Implementer.owl))
Send(SD_Agent, inform, requires (path planning))
Send(SD_Agent, inform, requires (simulation))

After receiving these messages from the service provider agent, service directory agent publishes the service in the Oracle UDDI registry.

6.2 User Agent ←→ Virtual Enterprise Agent

In this interaction, the user agent sends the input requirements to the virtual enterprise agent. Below are some sample input requirements to the VE agent:

Send (VE_Agent, inform, domain (Micro_Assembly)),
Send (VE_Agent, inform, input ()),
Send (VE_Agent, inform, radius (pin1, 0.5)),
Send (VE_Agent, inform, radius (pin2, 0.5)),
Send (VE_Agent, inform, radius (pin3, 0.5)),
Send (VE_Agent, inform, radius (cam1, 0.6)),
Send (VE_Agent, inform, radius (cam2, 0.6)),
Send (VE_Agent, inform, radius (cam3, 0.6)),
Send (VE_Agent, inform, goal ()),
Send (VE_Agent, inform, on (cam1, pin1)),
Send (VE_Agent, inform, on (cam2, pin2)),
Send (VE_Agent, inform, on (cam3, pin3))

This sequence of message states that the user would like to assemble three pins (*pin1*, *pin2*, *pin3*) of radius 0.5 into three cams of radius 0.6 by placing *pin1* on *cam1*, *pin2* on *cam2*, and *pin3* on *cam3*.

6.3 Virtual Enterprise Agent ←→ Ontology Agent

For the VE agent to process users' request, it needs to create a plan for doing it and who can provide the necessary services required to execute this plan. This information is available in the meta-information managed by the Ontology agent. The VE agent first queries the Ontology agent for meta-information about the services available in the system and devises a plan to achieve the goals of the users (as done in [10]).

In our experimental scenario, ontology for the Micro Assembly domain is developed and deployed in a Tomcat Application Server (refer to **Figures** 2 and 3). Some sample 3APL messages for this interaction are given below

Send (Ontology_Agent, inform, queryForMeta (Micro_Assembly))
Send (Ontology_Agent, inform, whatis (pin1))
Send (Ontology_Agent, inform, whatis (cam1))

Once the Ontology Agent receives the input from the VE agent, the Ontology Agent processes the input to find the corresponding ontology (in this case the ontology of Micro Assembly domain) and queries the ontology to find possible relationships between the input and the concepts it contained using the ontology plug-in. For sample input messages from VE agent, the ontology agent responds by sending the following messages,

Send (VE_Agent, inform, metaInfo (Micro_Assembly))
Send (VE_Agent, inform, steps ())
Send (VE_Agent, inform, physical_implementation ())
Send (VE_Agent, inform, planning ())
Send (VE_Agent, inform, simulation ())
Send (VE_Agent, inform, isObject (pin1, true))
Send (VE_ Agent, inform, isObject (cam1, true))

6.3 Virtual Enterprise Agent ←→ Service Directory Agent

With the meta information and the original input, the VE agent now requests the service directory agent for service providers. The sample messages of this interaction are given below.

Send (SD_Agent, inform, serviceProviderfor (physical_implementation))
Send (SD_Agent, inform, serviceProviderfor (planning))
Send (SD_Agent, inform, serviceProviderfor (simulation))

After receiving these messages, the service directory agent searches the UDDI registry for available service providers. In a UDDI registry, there may be more than one service provider who can serve the user agent's input request. Those service providers are known as potential partners in VE context. From the list of potential service providers, the service directory agent should choose one best service provider for the user agent. Before the selection of a best service provider, the Service directory agent will check for the requirements for each of the potential service providers. The requirements for a service provider may be correct inputs or even some services from other service providers. If all the requirements of a service provider are satisfied and it also satisfies the requirements of user agent, the service directory agent will announce the service provider as best partner. If user agent's requirement does not match with the service providers' requirements, then service directory agent will announce the unavailability of service providers. After finding the service providers, the service directory agent returns the access point URLs of each of the identified business vendors to the VE agent. Message transfers during this interaction are

Send (VE_Agent, inform,
    accessPointURL (http://128.123.245.156:9090/ontology/Implementer.owl)),
Send (VE_Agent, inform,
    accessPointURL (http://128.123.245.156:9090/ontology/planning.owl)),
Send (VE_Agent, inform,
    accessPointURL (http://128.123.245.156:9090/ontology/simulator.owl)),

The resulting access point URLs are then sent to User Agent for execution.

6.5 Service Directory Agent ←→ User Agent

After obtaining the access point URLs of service provider agents, the User agent executes the services available at the service provider sites.

7 Conclusion and Future Work

In this paper, a collaborative system is developed to form a Virtual Enterprise for the domain of Micro Assembly. 3APL language is used to develop the agents which constitute the collaborative system. Ontology for Micro Assembly domain is developed to provide a common ground to share the information contained in it among the agents. Although it is still an ad-hoc development, this prototypical system demonstrates that agent technologies can be very useful in VE development, a rather new area to agent researchers. In the future, we would like to study and develop methodologies for a systematic development of VE in the Micro Assembly domain.

References

1. Bandara, A., Payne, T., Roure, D., Clemo, G., An Ontological Framework for Semantic Description of Devices, *ISWC 2004*, Poster Session, Hiroshima, Japan, 7 - 11 Nov 2004.
2. Borgo, S., P. Leitão, The Role of Foundational Ontologies in Manufacturing Domain Applications, R. Meersman, Z. Tari et al. (eds.) OTM Confederated International Conferences, ODBASE 2004, Ayia Napa, Cyprus, 2004, LNCS 3290, pp. 670-688.
3. Camarinha-Matos, L. M., Asfarmanesh, H., Virtual Enterprise Modeling and Support Infrastructures: Applying Multi-Agent System Approaches in *Multi- agent Systems and Applications*, in LNAI 2086, Springer, July 2001.
4. Hardwick, M., Spooner, D. L., Rando, T., and Morris, K. C. 1996. Sharing manufacturing information in virtual enterprises. *Commun. ACM* 39, 2 (Feb. 1996), 46-54. http://doi.acm.org/10.1145/230798.230803
5. Hindriks, K. V., De Boer, F. S., Van Der Hoek, W., and Meyer, J.-J. Ch. Agent Programming in 3APL, *Autonomous Agents and Multi-Agent Systems, ACM*, 2:4, 357–401, 1999.
6. Genesereth, M. R. and Fikes, R. E. Knowledge Interchange Format (KIF) Version 3.0, *Reference Manual*.
7. Munindar P. Singh. Agent Communication Languages: Rethinking the Principles, *Computer*, vol. 31, no. 12, pp. 40-47, December, 1998.
8. Petersen, S. A., Gruninger, M., An Agent-based Model to Support the Formation of Virtual Enterprises, *Int. ICSC Symposium on Mobile Agents and Multi-Agent in Virtual Organizations and E-Commerce (MAMA '2000)*, in Australia, 11-13 Dec. 2000.
9. Petersen, S. A., Rao, J., Matskin, M., AGORA Multi-agent Architecture for Implementing Virtual Enterprises, *Norsk Informatikkonferanse* NIK2003, Oslo, Norway, 2003.
10. McIlraith, S., Son, T. C., and Zeng, H. Semantic Web Services, *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46-53, March/April, 2001.
11. The OWL Services Coalition, "OWL-S: Semantic Markup for Web Services", http://www.daml.org/services/owl-s/1.0/owl-s.html.
12. Wilbur, S., Computer Support for Co-operative Teams: Applications in Concurrent Engineering, *IEEE Colloqium on Current Development in Concurrent Engineering Methodologies and Tools*, June 1994.
13. M. Grüninger and C. Menzel. The Process Specification Language (PSL) Theory and Applications, *AAAI Magazi*, 63-74, *Fall 2000*.

A Modelling Framework for Generic Agent Interaction Protocols

José Ghislain QUENUM[2], Samir AKNINE[1], Jean-Pierre BRIOT[1], and Shinichi HONIDEN[2]

[1] Laboratoire d'Informatique de Paris 6,
8 rue du Capitaine Scott, 75015 Paris, France
[2] National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan

Abstract. This paper presents a framework to represent generic protocols. We call generic protocols, agent interaction protocols where only a general behaviour of the interacting entities can be provided. Our framework is grounded on the AUML graphical formalism. From this formalism, we identified five fundamental concepts on top of which we defined the formal specifications for the framework. We address a lack in protocol representation by emphasising the description of actions performed in the course of interactions based on generic protocols. The framework is formal, expressive and of practical use. It helps decouple interaction concerns from the rest of agent architecture. Several application levels exist for our framework. First, we used it to address two issues faced in the design of agent interactions based on generic protocols. At a more concrete level, this framework can be used to publish the protocols agent interactions are based on in a multi-agent system.

1 INTRODUCTION

Interaction is one of the key aspects in agent-oriented design. It allows agents to put together the necessary actions in order to perform complex tasks collaboratively. The coordination mechanism needed for a safe execution of these actions is often governed by a sequence of message exchanges: interaction protocols. Usually, only a general description of how agents should behave during the interactions is provided. Such protocols are called generic protocols. An issue in open and heterogeneous multi-agent systems (MAS) is concerned with the description of generic protocols, especially with respect to their correct interpretation. A subsequent issue is the need to decouple interaction concerns from the rest of agent architecture.

To date, there has been some endeavour to develop new protocol representation formalisms. The formalisms developed thus far have several drawbacks. They usually focus on data exchange through a communication channel (Promela/SPIN [7]). Some others are either informal (or semi-formal) (e.g., AUML [1]) or demand advanced knowledge in logics (e.g., the formal framework in [10]). Therefore, there is an obvious need for a formal, yet practical and expressive generic protocol representation framework. Additionally, such a framework should provide the building blocks to help decouple the interaction concerns from the rest of agent architecture. We address this need in this paper.

The solution we arrived at is a framework for the description of generic protocols. It complies with most of the criteria required of a conversation policy in [6]. The philosophy of our framework is to start from AUML, which is a well established agent interaction representation formalism. But we depart from AUML by addressing the lacks and incompleteness which limit it. A common trend in protocol representation consists of describing only the sequence of message exchanges. However, some actions are needed to produce these messages and handle them when received. As we will see later, some actions might be executed beyond the communication level during an interaction. Thus, in addition to the description of message exchange, our framework introduces the description of actions needed in the course of an interaction. This provides us with the ability of describing the behaviour agents will exhibit while playing a role in a protocol. A particular aspect in our framework is our focus on generic protocols. This keeps us from providing a complete representation for actions. We introduced action categories to fix this weakness.

Our framework offers several advantages. It builds on the graphical representation in AUML, which eases the message exchange perception for human designers. In addition, it offers the means to depict what happens beyond the message exchange level. The framework is expressive, formal and of practical use for protocol representation. Particularly, we offer at least the same expressiveness as in AUML (and its extensions) without introducing new control flows. Rather, we only use event description and (if necessary) three connectors: *and*, *or* and *xor*. We also ease the implementation of protocols in our framework by providing a XML representation. As an application, we used our framework to address two issues in agent interaction design of open and heterogeneous MAS: (1) an automatic derivation of agent interaction model from generic protocol specifications, in order to address the issue of consistency during interactions based on generic protocols in an heterogeneous MAS; and (2) an analysis of generic protocol specifications in order to enable agents to dynamically select protocols when they have to perform a task in collaboration. A more practical usage of our framework is the possibility to publish protocol specifications for agent interactions in a MAS.

The remainder of this paper is organised as follows. Section 2 discusses some related work. Section 3 introduces the fundamental concepts we use in the framework and presents both the specifications and their semantics. Section 4 discusses some properties one can check for a protocol represented following this framework. Finally, section 5 concludes the paper.

2 RELATED WORK

Several formalisms have been developed to represent interaction protocols. In this section, we discuss the most suitable ones for agent interaction.

AUML [1] and its extensions are graphical frameworks for protocol diagram representation. These frameworks, though practical and easy to use, are informal (or semi formal). It is then hard to check properties or even define the semantics of a protocol represented in these formalisms. [15] and [2] automate the translation process from

AUML to a textual description, which is more machine readable. The advantage of this automatic translation, though undebatable, is weakened by many other AUML original limitations, f.i., the lack of emphasis on action representation in protocol representation.

Some formal frameworks have been proposed for protocol representation. [14] defined a framework using concepts similar to ours. However, in our case agents are not represented in protocol specifications. They are expected to play (in a protocol configuration and instantiation standpoint) roles at runtime. [10] made significant advances in the area of protocol representation for agent interaction. This work developed a formal framework which combines Propositional Dynamic Logic and belief and intention modalities (PDL-BI). The framework covers a broad spectrum of issues related to agent interactions. However, it requires advanced knowledge in logics. In our opinion, logics is useful to define the semantics and check some properties for protocols. But due to the complexity it may introduce, we believe that it should be hidden at the specifications stage, as usually done in programming languages. Additionally, PDL-BI focuses on the messages exchange. But, as we showed above, agent interaction protocols demand more than message exchange.

IOM/T [3] is a more recent language for interaction representation. Our work, though sharing some similarities with IOM/T, departs from it on the following points. Firstly, we focus on generic protocols, where we consider generic actions. Secondly, the behaviour of the agents in IOMT/T (the actions they perform) is not associated with the events which occur in the MAS. Thirdly, the language is Java-like. However, we believe that a protocol description language is supposedly a declarative one. Especially for open and heterogeneous MAS. We address this need in this paper by developing a formal framework for generic protocols representation. Our framework is a declarative language and offers expressiveness as well as ease of use.

3 THE FRAMEWORK

We introduce the fundamental concepts our framework is based on. Then, we present the specifications and the semantics of these concepts.

3.1 FUNDAMENTAL CONCEPTS

Our framework is based on the AUML protocol diagram. Thereof, we identified five fundamental concepts: *protocol*, *role*, *event*, *action* and *phase*. A graphical illustration of these concepts is given in Fig. 1.

Definition 1. Protocol

A protocol is a sequence of message exchanges between at least two roles. The exchanged messages are described following an Agent Communication Language (ACL) e.g., FIPA ACL [5], KQML [9].

More formally, a protocol consists of a collection of roles $\mathbf{R}$, which interact with one another through message exchange. The messages belong to a collection $\mathbf{M}$ and the exchange takes place following a sequence, $\mathbf{\Omega}$. A protocol also has some intrinsic properties $\mathbf{\Theta}$ (attributes and keywords) which are propositional contents that provide

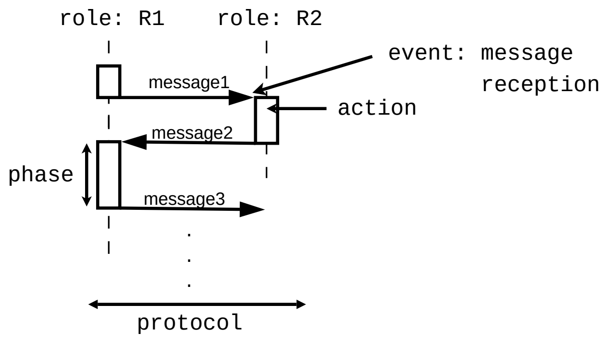


Fig. 1. Graphical illustration of concepts in generic protocols.

a context for further interpretation of the protocol. We note $\mathbf{p} \overset{\text{def}}{=} < \mathbf{\Theta}, \mathbf{R}, \mathbf{M}, \mathbf{\Omega} >$. In $\mathbf{\Omega}$, the message exchange sequence, each element is denoted by $r_\imath \xrightarrow[a_\alpha, m_{k-1}]{m_k} r_\jmath$, to be interpreted as "the role $r_\imath$ sends the message m_k to $r_\jmath$, and that m_k is generated after action a_α's execution and the prior exchange of m_{k-1}". Additional elements may be introduced in this representation, but we do not discuss them in this paper.

Definition 2. Generic Protocol
A generic protocol is a protocol wherein the actions which are taken, to handle, produce the contents of exchanged messages, etc. cannot be thoroughly described. A complete description of these actions depends on the architecture of each agent playing a role in the protocol.

Note that the attributes and keywords we use in the current version have been identified from our experimentations. Currently, we only use three attributes: *class*, *return_value* and *participants_count*. *class* is the type of processing performed through the execution of an interaction based on this protocol (e.g., we use *request* to denote that the participant performs some task on behalf of the initiator). *return_value*, when any, depicts how the final result is represented. *participants_count* is the number of distinct participant roles in the protocol. As for keywords, several ones can be used in the current version. For example, *IncrementalProcess* means that some partial results may be considered for the ongoing process. Some more experiments are needed to extend these attributes and keywords.

Each of the communicating entities is called a role. Roles are understood as standardised patterns of behaviour required of all agents playing a part in a given functional relationship in the context of an organisation [4].

Definition 3. Role *In our framework, a role consists of a collection of phases. As we will see later in Section 3.2, a role may also have global actions (which are executed outside all phases) and some data other than message content,* variables.

$\forall \mathbf{r} \in \mathbf{R}, \mathbf{r} \overset{\text{def}}{=} < \mathbf{\Theta_r}, \mathbf{P_h}, \mathbf{A_g}, \mathbf{V} >$, where $\mathbf{\Theta_r}$ corresponds to the role's intrinsic properties (e.g., cardinality) which are propositional contents that help further interpret the role, $\mathbf{P_h}$ the set of phases, $\mathbf{A_g}$ the set of global actions and $\mathbf{V}$ the set of variables. The roles can be of two types: (1) *initiator*, the unique role of the protocol in charge of starting its execution; (2) *participant*, any role partaking in an interaction based on the protocol.

The behaviour of a role is governed by events. An event is an atomic change which occurs in the environment of the MAS. An informal description of the types of events we consider in our framework is given in Table 1. A more formal interpretation of these events is discussed in Section 3.3. The behaviour a role adopts once an event occurs is described through actions.

Event Type	**Description**
Change	The content of a variable has been changed.
Endphase	The current phase has completed.
Endprotocol	The end of the protocol is reached.
Messagecontent	The content of a message has been constructed.
Reception	A new message has been received.
Variablecontent	The content of a variable has been constructed.
Custom	Particular event (error control or causality).

Table 1. Event Types.

Definition 4. Action
An action is an operation a role performs while executing. This operation transforms the whole environment or the internal state of the agent currently playing this role. An action has a category ν, a signature Σ and a set of events it reacts to or produces. We note $a \overset{def}{=} < \nu, \Sigma, E >$.

Since our framework focuses on generic protocols, we can only provide a general description for the actions which are executed in these protocols. Hence, we introduced action categories to ease the definition of a semantics for these actions. Table 2 contains an informal description of these categories. We discuss their semantics in Section 3.3.

Definition 5. Phase *Some successive actions sharing direct links can be grouped together. Each group is called a phase. Two actions share a direct link if the (or only a part of them) input arguments of one are generated by the other (f.i. when a* send *action sends the message generated in a prior action).*

3.2 FORMAL SPECIFICATIONS

The formal specifications are defined through a EBNF grammar. Only essential parts of this grammar are discussed in this section. A thorough description of this grammar is given in Appendix A. In sake of easy implementation of generic protocols, we represent

Action Type	**Description**
Append	Adds a value to a collection.
Remove	Removes a value from a collection.
Send	Sends a newly generated message.
Set	Sets a value to a variable.
Update	Updates the value of a variable.
Compute	Computes a new information.

Table 2. Action categories.

them in XML in our framework. However, as XML is too verbose, a simpler (bracket-based) representation will be used for illustration in this paper.

RUNNING EXAMPLE We will use the Contract Net Protocol (CNP) [13] to illustrate the specifications we present. The sequence diagram (protocol diagram in AUML) of this protocol is given in Fig. 2. The labels placed on message exchange arrows in the figure are not performatives, but message identifiers.

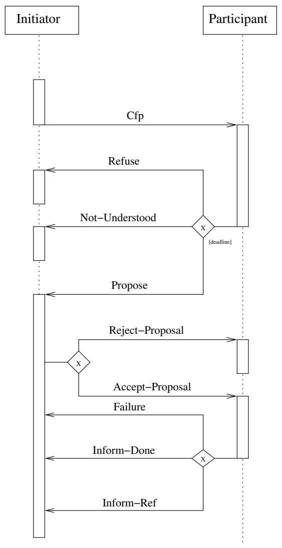


Fig. 2. The Contract Net Protocol.

The rationale of the CNP consists of an initiator having some participants perform some processing on its behalf. But beforehand, the participants which will perform the processing are selected on the basis of the bids they proposed, in-reply to the initiator's call for proposals. When the selected participants are done with their processing, each of them notifies the initiator agent of the correct execution (or error occurrence) of the part it committed in performing.

PROTOCOL The following production rules define a protocol:
<protocol>:=<protproperties><roles><messagepatterns>
<protproperties>:=<protdescriptors><protattributes><protkeywords?>
<protdescriptors>:=<identifier><title><location>
<protattributes>:=<class><participantcount><return>
<protkeywords>:=<protkeyword+> <protkeyword>:="IncrementalProcess"|...
An illustration of these rules is given as follows.

```
(protocol
 (protocolproperties
  (protocoldesc ident='cnpprot' title='ContractNet' location='KqmlCnp.xml')
  (protocolattr class='Request' participantcount='1' return='operationresult)
  (protkeyws '...'))
 (roles ...)
 (messagepatterns ...))
```

As one can see from these rules, the exchange sequence $\mathbf{\Omega}$ is not explicitly specified. Actually, it is described through *send* actions in each role. When several messages can be sent, we use connectors *and*, *or* and *xor* to compose them.

ROLE Protocol diagrams only show the communication flow between roles. However, there may be some information beyond the communication level. For example, in the CNP, the action an initiator executes in order to make a decision upon the participants' bids is hidden behind the communication flow. Actually, this action exploits information from different participants of the protocol. Moreover, information like the deadline for bidding, cannot be extracted from any message content. Then, we introduced a global area for each role where we describe actions which are beyond the communication flow, as well as data which cannot be extracted from any message content. Note that actions relevant to the global area are no more associated with any phase. The production rules hereafter define a role.
<roles>:=<role><role>|<roles><role>
<role>:=<roleproperties><variables?><actions?><phases>
<roleproperties>:=<roledescriptors><roleattributes><rolekeywords?>
<roledescriptors>:=<identifier><name>
<roleattributes>:=<cardinality>
<variables>:=<variable+> <variable>:=<ident><type>
<actions>:=<action+>
Each role is described through its intrinsic properties (f.i., name and cardinality), its variables (pieces of information the role handles which are not extracted from any message content), its global actions and the phases the non global actions are grouped in.

In the example below, the *initiator* role of the CNP has three variables: `deadline`, `bidsCol` and `deliberations`. `deadline` contains the time when bidding should stop. `bidsCol` is a collection where participants' bids are stored. `deliberations` contains the decision (accept or reject) the initiator made upon each bid. Each variable has an identifier and the type of the data it contains. The content of a variable is characterised using some abstract data types. We also use these data types to represent message content and action signature. String, Number and Char are some examples of the data types we use in our framework. The description of these types is out of the scope of this paper. The only global action in this role is named `Deliberate`. Through this action, the initiator makes a decision upon the participants' bids. Global actions are described in the same way like local (located in a phase) ones: category, signature and events. The description of `Deliberate` explains itself from the example. The special word *eventref* is used here to refer to an event defined elsewhere (*change* event which occurred against the `bidsCol` variable). As we will see later, this word sometimes introduces causality between actions.

```
(role ident='initiator'
 (roleproperties (roledescriptors ident='initiator' name='Initiator')
  (roleattributes cardinality='1'))
 (variables (variable ident='bidsCol' type='collection')
  (variable ident='deliberations' type='map')
  (variable ident='deadline' type='date'))
 (actions(action category='compute' description='Deliberate'
   (signature (arg type='date' dir='in')
    (arg type='collection' dir='in')(arg type='map' dir='out'))
   (events (event type='change' dir='in' object='deadline' ident='evt0')
    (eventref dir='in' ident='evt5')
    (event type='change' dir='out' object='deliberations' ident='evt1'))))
  (phases ...))
```

PHASE As stated above, each phase is a sequence of actions that share some direct links. We use the following rules to define a phase:
<phases>:=<phase+>
<phase>:=<actions>
<action>:=<category><description?><signature><events>
For example, in the initiator role of the CNP, the first phase consists of producing and sending the `cfp` message. This phase contains two actions: `prepareCFP` and `sendCFP`. `prepareCFP` produces the `cfp` message. It is followed by `sendCFP` which sends the message to each identified participant.

```
(phase ident='phs1'
 (actions (action category='compute' description='prepareCFP'
   (signature(arg type='date' dir='in')(arg type='any' dir='out'))
   (events (event type='variablecontent' dir='in' object='deadline')
    (event type='messagecontent' dir='out' object='cfp' ident='evt2')))
  (action category='send' description='sendCFP'
   (signature (message ident='cfp'))
   (events(eventref dir='in' ident='evt2')
   (eventref type='custom' dir='out' ident='cus01')
    (event type='endphase' dir='out' ident='evt3')))))
```

MESSAGE Though we did not define messages as a concept, we use them in the formal specifications because they contain part of the information manipulated during interactions. The concept of message is well known in ACL, and their semantics is defined accordingly.

We propose an abstract representation of messages, which we call *message patterns*. A message pattern is composed of the performative and the content type of the message. We also offer the possibility to define the content pattern, a UNIX-like regular expression which depicts the shape of the content. Note that at runtime, these messages will be represented with all the fields as required by the adopted ACL. In our framework, we represent all the message patterns once in a block and refer to them in the course of the interaction when needed. In our opinion, it sounds to constrain to the use of only one ACL all along a single protocol description. The following rules define message patterns.
<messagepatterns>:=<acl><messagepattern+>
<acl>:='fipa'|'kqml'
<messagepattern>:=<performative><identifier><content>
<content>:=<type><pattern?>
The example below describes the message patterns used in the CNP.

```
(messagepatterns acl='Kqml'
 (messagepattern performative='achieve' ident='achmsg'
  (content type='any' pattern='...'))
 (messagepattern performative='sorry' ident='refuse'
  (content type='null' pattern='...'))
 (messagepattern performative='tell' ident='propose'
  (content type='any' pattern='...'))
 (messagepattern performative='deny' ident='reject'
  (content type='null' pattern='...'))
 (messagepattern performative='tell' ident='accept'
  (content type='string' pattern='...')) ...)
```

DESIGN GUIDELINE As a guideline for protocol design and description in our framework, we recommend several design rules. They guarantee the correctness of a protocol represented in our framework. We introduce some of them here.

Proposition 1. *For each role of a protocol, there should be at least one action which drives into the terminal state. Every such action should be reachable from the role's initial state.*

Corollary 1. *From their semantics, roles can be represented as graphs. And for every path in this graph, there should be an action which drives to a terminal state.*

Proposition 2. *When two distinct transitions can be fired from a state, the set of events which fire each one of the transitions, though intersect-able, should be distinguishable.*

Proposition 3. *When an action produces a message, it should be immediately followed by a* send *action, which will be responsible for sending the message.*

3.3 SEMANTICS OF THE CONCEPTS

EVENT As we saw, an event informs of an atomic change. It may have to do with the notified role's internal state. But usually, the notification is about other roles' internal state. Therefore, events are the grounds for roles coordination. Due to space constraints, we only discuss the semantics of two types of events in this section.

change: this event type notifies of a change of the value of a variable. Let v be this variable, *change(*v*)* denotes this event. In order to define the semantics of our concepts, we introduce some expressions in a meta-language, which we call primitives. These primitives are functions and predicates. *value* is one of these primitives (actually a function). It returns the value of a data at a given time. Let $\mathcal{T}$ be the time space, and d and t a data and a time respectively ($t \in \mathcal{T}$), $Value(d, t)$ denotes this function. $\text{Value}(d, t) = \emptyset$ means that the data d does not exist yet at time t. We interpret the *change* event as follows: $\exists\ (t_1, t_2) \in \mathcal{T} \times \mathcal{T}$:

$$(t_1 \neq t_2) \wedge (\text{Value}(v, t_1) \neq \emptyset) \wedge (\text{Value}(v, t_1) \neq \text{Value}(v, t_2))$$

endprotocol: this event type notifies of the end of the current interaction. For each role, all the phases have either completed or are unreachable. Also any global action of each role is either already executed or unreachable. A phase is unreachable if none of its actions is reachable. Actually, if the initial action is unreachable, the phase it belongs to will also be unreachable. Again, we introduced three new primitives: *Follow*, *Executed* and *Unreachable*. *Follow* is a function which returns all the immediate successors of a phase. Let p_1 and p_2 be two phases, p_1 immediately follows p_2, if any of the input events of the initial action of p_1 refers to a prior event generated by one of the actions (usually the last one) of p_2. *Unreachable* is a predicate which means that the required conditions for the execution of an action do not hold. Therefore, this action cannot be executed. Finally, *Executed* is a predicate which means that an action has already been executed. Let P_r be the set of phases and $A_{p_{kr}}$ the set of executable actions for phase p_{kr} in a role r. Let also A_{G_r} be the set of global actions for role r. We interpret the *endprotocol* event as follows:$\forall r \in \mathcal{R}, \forall a_\alpha \in A_{G_r}$,

$$(\text{Unreachable}(a_\alpha) \vee \text{Executed}(a_\alpha)) \wedge (\forall p_{kr} \in P_r,\ (\text{Follow}(p_{kr}) = \emptyset) \vee (\forall a_i \in A_{p_{kr}}, \text{Unreachable}(a_i)))$$

ACTION Actions are executed when events occur. And once executed, they may produce some new change in the MAS. Events are therefore considered as *Pre* and *Post* conditions for actions' execution. Here again, we only discuss the semantics of the *append* and *send* action categories.

Let $\mathcal{E}$ be the set of all the event types we consider in our framework. We define $\mathcal{E}'$ as a subset of $\mathcal{E}$: $\mathcal{E}' = \mathcal{E} - \{endphase, endprotocol\}$.

append: this action adds a data to a collection. Let a_i be such an action,

$$\begin{aligned}
&\text{Pre} = \textstyle\bigvee_j e_j, \text{where } e_j \in \mathcal{E}' \\
&\text{Post} = \textstyle\bigvee_j e_j, \exists k\ e_k =' change' \wedge (\exists (t_1, t_2) \in \mathcal{T} \times \mathcal{T}, \exists d, v \in args(a_i), \\
&(t_1 < t_2) \wedge (\text{isElement}(v, d, t_1) = false) \wedge (\text{isElement}(v, d, t_2)))
\end{aligned}$$

isElement() is a predicate which returns true when a data belongs to a collection at a given time. *args()* returns the arguments of an action.

send: this action sends a message. It is effective both at the sender and the receiver sides. Let a_i be such an action. We interpret it as follows: at the sender side:

Pre $= \bigvee_j e_j$, where $\forall m_j \in arguments(a_i), \exists\, k,\, e_k =$ messagecontent(m_j)
Post $= (Trans(m_j) = true)$

at the receiver side:

Pre $= \emptyset$
Post $= \bigvee_j e'_j$, where $\forall m_j \in arguments(a_i), \exists!\, k,\, e'_k =$ reception(m_j)

ACL usually define the semantics of their performatives by considering the belief and intention of the agents exchanging (sender and receiver) these performatives. This approach is useful to show the effect of a message exchange both at the sender and the receiver sides. In our framework, we adopt a similar approach when an action produces or handles a message. We use the knowledge the agent performing this action has with respect to the message. Hence, we introduce a new predicate, $Know(\phi, a_g)$, which we set to true when the agent a_g has the knowledge ϕ. $Know$ is added to the post conditions of the action when the latter produces a message. It is rather added to the pre conditions of the action when it handles a message. Note that ϕ is the (propositional) content of the message. Moreover, when an action ends up a phase or the whole protocol, its Post condition is extended with the *endphase* and *endprotocol* events respectively.

PHASE The semantics of a phase is that of a collection of actions sharing some causality relation. The direct links between actions of a phase are augmented with a causality relation introduced by events. We note $\mathbf{p_h} \overset{\text{def}}{=} < \mathbf{A}, \prec >$, where $\mathbf{A}$ is a set of actions and $\prec$ a causality relation which we define as follows ($|\mathbf{A}|$ is the cardinality of $\mathbf{A}$):
$\forall \mathbf{a}_\imath, \mathbf{a}_\jmath \in \mathbf{A}, \mathbf{a}_\imath \neq \mathbf{a}_\jmath, \mathbf{a}_\imath \prec \mathbf{a}_\jmath \Longleftrightarrow |\mathbf{A}| > 1 \,\wedge\, \exists e \in Post(\mathbf{a}_\imath), e \in Pre(\mathbf{a}_\jmath)$.

Proposition 4. *Let* $\mathbf{a}_\imath$ *and* $\mathbf{a}_\jmath$ *be elements of* $\mathbf{P_h}$*, such that* $\mathbf{a}_\imath$ *always precedes* $\mathbf{a}_\jmath$*,*

$$(\mathbf{a}_\imath \prec \mathbf{a}_\jmath) \vee (\exists \mathbf{a_p}, \ldots, \mathbf{a_k}, \mathbf{a}_\imath \prec \mathbf{a_p} \ldots \prec \mathbf{a_k} \prec \mathbf{a}_\jmath)$$

ROLE An event generated at the end of a phase can be referred to in other phases. Thus, the causality relation between actions of phases can be extended to interpret roles. We consider a role as a labelled transition system having some intrinsic properties. $\mathbf{r} \overset{\text{def}}{=} < \mathbf{\Theta_r}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow >$ where:

– $\mathbf{\Theta_r}$ are the intrinsic properties of the role;
– $\mathbf{S}$ is a finite set of states;
– $\mathbf{\Lambda}$ contains transitions labels. These are the actions the role performs while running;
– $\longrightarrow \subseteq \mathbf{S} \times \mathbf{\Lambda} \times \mathbf{S}$ is a transition function.

As an illustration, we give part of the semantics of the initiator role of the CNP, which we call $\mathbf{r_0}$. $\mathbf{r_0} = < \mathbf{\Theta_{r_0}}, \mathbf{S}, \mathbf{\Lambda}, \longrightarrow >$, with:

– $\mathbf{\Theta_{r_0}} =' cardinality = 1 \,\wedge\, isInitiator = true\ \ldots'$;

- $\mathbf{S} = \{\mathbf{S_0}, \mathbf{S_1}, \mathbf{S_2}, \mathbf{S_3}, \mathbf{S_4}, \mathbf{S_5}, \mathbf{S_6}, \mathbf{S_7}, \mathbf{S_8}, \mathbf{S_9}, \mathbf{S_{10}}, \mathbf{S_{11}}\}$;
- $\mathbf{\Lambda} = \{\mathbf{a_0}, \mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \mathbf{a_4}, \mathbf{a_5}, \mathbf{a_6}, \mathbf{a_7}\}$;
- $\longrightarrow = \{(\mathbf{S_0}, \mathbf{a_0}, \mathbf{S_1}), (\mathbf{S_1}, \mathbf{send_{[m_0]}}, \mathbf{S_2}), (\mathbf{S_2}, \mathbf{a_1}, \mathbf{S_7}), (\mathbf{S_2}, \mathbf{a_2}, \mathbf{S_3}), (\mathbf{S_3}, \mathbf{a_4}, \mathbf{S_4}), (\mathbf{S_4}, \mathbf{a_3}, \mathbf{S_5}),$ $(\mathbf{S_5}, \mathbf{send_{[m_3]}}, \mathbf{S_{11}}), (\mathbf{S_5}, \mathbf{send_{[m_4]}}, \mathbf{S_6}), (\mathbf{S_6}, \mathbf{a_5}, \mathbf{S_8}), (\mathbf{S_6}, \mathbf{a_6}, \mathbf{S_9}), (\mathbf{S_6}, \mathbf{a_7}, \mathbf{S_{10}})\}$

m_0, m_3 and m_4 belong to the set of messages exchanged during the protocol.

PROTOCOL The semantics of a protocol is a combination of the semantics of its intrinsic properties, that of each role and finally the semantics of the coordination mechanism. Recall that the coordination mechanism, in our case, is the sequence of message exchanges. The sequence of messages actually exchanged during the interaction is known only at runtime. This raises up one of the limitations of the work concerned with agent interaction protocols semantics. They usually proposed *a priori* semantics for protocols. However, as protocols generally offer several possible exchange sequences, several possible semantics may coexist for an interaction based on a protocol. [8] proposed *a posteriori* semantics through a platform called *Protocol Operational Semantics (POS)*. We build on this platform and adopt *a posteriori* semantics for protocols in our framework. Two main reasons account for such an approach. Firstly, the messages exchange sequence can be mapped to a graph of possibilities for exchanged messages. Therefore, the semantics of an interaction based on this protocol consists of a path in this graph. Secondly, the semantics of communicative acts defined in ACL is not enough to define the semantics of a protocol. The executed actions' semantics should also be included. However, apart from send actions, all the other actions can only have general characterisation before the execution of the interaction. A more precise semantics of these actions can only be known at runtime.

As an illustration, let us assume that the semantics of each role of the CNP is known, we define that of the whole protocol as follows. $\mathbf{p} =< \mathbf{\Theta}, \mathbf{R}, \mathbf{M}, \mathbf{\Omega} >$, where $\mathbf{R} = \{\mathbf{r_0}, \mathbf{r_1}\}$ and $\mathbf{M} = \{\mathbf{m_0}, \mathbf{m_1}, \ldots \mathbf{m_7}\}$. $\mathbf{m_0}$ corresponds to `cfp`, $\mathbf{m_1}$ corresponds to `refuse`, etc. (see Fig. 2). $\mathbf{\Omega} = \mathbf{\Omega_1}|\mathbf{\Omega_2}|\mathbf{\Omega_3}|\mathbf{\Omega_4}|\mathbf{\Omega_5}|\mathbf{\Omega_5}|\mathbf{\Omega_6}$. $\mathbf{\Omega_6}$ is the case where everything went correctly and the participant notifies the initiator of the correct performance of the task. $\mathbf{\Omega_6} =< r_0 \xrightarrow[a_0]{m_0} r_1, r_1 \xrightarrow[a_8, m_0]{m_2} r_0, r_0 \xrightarrow[a_3, m_2]{m_3} r_1, r_1 \xrightarrow[a_{10}, m_4]{m_7} r_0 >$

4 PROPERTIES

4.1 LIVENESS

Proposition 5. *For every role of a protocol, events will always occur and fire some transition until the concerned role enters a terminal state.*

Proof. Each role is a transition system. And from the description of transition systems, unless an error occurs, an event will always occur and require to fire a transition until the role enters a terminal state, where the execution stops.

4.2 SAFETY

We consider two safety properties: *consistent messages exchange* and *Unambiguous protocol execution*.

Proposition 6. Consistent messages exchange *Messages exchange is consistent in our framework. Precisely, any message a role sends is received and handled at least by one role. By the same token, any message a role receives has a sender (generally another role).*

Proof. We prove both parts of the proposition.

1. The sequence of message exchanges $\mathbf{\Omega}$ is described as follows: $\mathbf{\Omega} =< r_0 \xrightarrow[a_0]{m_0} r_1, r_1 \xrightarrow[a_8, m_0]{m_2} r_0, \ldots >$. This representation shows that any message sent is received and handled by at least one role.
2. Any received message has been generated elsewhere, since its identifier exists. Additionally, from proposition 3, any message generated is automatically sent. Hence, any message received is sent by a role.

Proposition 7. Unambiguous protocol execution *For each action a role can take, there is an unambiguous set of events which fire its execution.*

Proof. The proof follows from the direct application of proposition 2 and is omitted here because of space constraints.

4.3 TERMINATION

Proposition 8. *Each role of a protocol represented in our framework always terminates.*

Proof. From Proposition 1, each role has at least an action which brings that role to a terminal state. Once this terminal state is reached, the interaction stops for the concerned role. When all the roles enter a terminal state, the whole interaction definitely stops.

This proof is insufficient when there are several alternatives or loops in the protocol. Corollary 1 addresses this case. Actually, only one path of the graph (with respect to the transition system) corresponding to the current role will be explored. And as this path ends up with an action driving to a terminal state, the role will terminate.

5 CONCLUSION

We believe that a special care is needed in representing generic protocols, since only partial information can be provided for them. Therefore, We developed a framework to represent generic protocols for agent interactions. Our framework puts forth the description of the actions performed by the agents during interactions, and hence highlights their behaviour during protocols execution. In this, we depart from the usual protocol representation formalisms which only focus on exchanged messages descriptions. Our framework is based on a graphical formalism, AUML. It is formal, at least as expressive as AUML (and its extensions) and of practical use. As we discussed in the paper, this framework has been used to address issues in agent interaction design.

Since actions in generic protocols can be described only in a general way, a more precise description of these actions is dependent on the architecture of the agent about to perform them in the context of an interaction. This is usually done by hand by agent designers when they have to configure agent interaction models. Doing such a configuration by hand may lead to inconsistent message exchange in an heterogeneous MAS. We address this issue by developing some mechanisms to automatically carry this configuration out. These mechanisms consist of looking for similarities between the functionalities from agent architecture and actions of protocols. These mechanisms are presented in [12].

Protocol selection is another issue we faced while designing agent interactions based on generic protocols. Usually, agent designers select the protocols their agents will use to interact during the performance of collaborative tasks. But this static protocol selection severely limits interaction in open and heterogeneous MAS. Thus, we developed some mechanisms to enable agents to dynamically select the protocol they will use to interact. These mechanisms require some reasoning about the specifications of the protocols. Again, we used this framework, since it enables us to reason about the mandatory coordination mechanisms for the performance of collaborative tasks. We described part of the mechanisms we proposed in [11].

References

1. B. Bauer and J. Odell. UML 2.0 and Agents: How to Build Agent-based Systems with the new UML Standard. *Journal of Engineering Applications of Artificial Intelligence*, 18:141–157, 2005.
2. G. Casella and V. Mascardi. From AUML to WS-BPEL. Technical report, Computer Science Department, University of Genova, Italy, 2001.
3. T. Doi, Y. Tahara, and S. Honiden. IOM/T: an Interaction Description Language for Multi-agent Systems. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 778–785, 2005.
4. M. Esteva, J. A. Rodriguez, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*. 2001.
5. FIPA. FIPA Communicative Act Library Specification. Technical report, Foundation for Intelligent Physical Agents, 2001.
6. M. Greaves, H. Holmback, and J. Bradshaw. Waht is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents 1999*, 1999.
7. G.J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.
8. J-L Koning and P-Y Oudeyer. Introduction to POS: A Protocol Operational Semantics. *International Journal on Cooperative Information Systems*, 10(1 2):101–123, 2001. Special Double Issue on Intelligent Information Agents: Theory and Applications.
9. Y. Labrou and T. Finin. A proposal for a new KQML Specification. Technical report, University of Maryland Baltimore County (UMBC), 1997.

10. S. Paurobally, J. Cunningham, and N. R. Jennings. A Formal Framework for Agent Interaction Semantics. In *Proceedings. 4th International Joint Conference on autonomous Agents and Multi-Agent Systems*, pages 91–98, Utrecht, The Netherlands, 2005.
11. J. G. Quenum and S. Aknine. A Dynamic Joint Protocols Selection Method to Perform Collaborative Tasks. In P. Petta M. Pechoucek and L.Z. Varga, editors, *4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS 2005)*, LNAI 3690, pages 11–20, Budapest, Hungary, September 2005. Springer Verlag.
12. J. G. Quenum, A. Slodzian, and S. Aknine. Automatic Derivation of Agent Interaction Model from Generic Interaction Protocols. In P. Giorgini, J. P. Muller, and J. Odell, editors, *Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering*. Springer Verlag, 2003.
13. G. Smith. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Trans. on Computers*, 29(12):1104–1113, 1980.
14. C. Walton. Multi-agent Dialogue Protocols. In *Proceedings of the Eight Int. Sympossium on Artificial Intelligence and Mathematics*, 2004.
15. M. Winikoff. Towards Making Agent UML Practical: A Textual Notation and Tool. In *Proc. of the First Int. Workshop on Integration of Software Engineering and Agent Technology (ISEAT)*, 2005.

A EBNF Grammar

```
  <protocol>:=<protproperties><roles><messagepatterns>
<protdescriptors>:=<protdescriptors><protattributes><protkeywords?>
<protdescriptors>:=<identifier><title><location>
<protattributes>:=<class><participantcount><return>
<protocolkeywords>:=<protocolkeyword+>
<protocolkeyword>:="containsconcurrentroles"|"iterativeprocess"|
                 "incrementalprocess"|"subscriptionrequired"|
                 "alterableservicecontent"|"alterableproposalcontent"
 |"dividableservice"
<title>:=<word>
<class>:=<word>
<location>:=<locationheader><path>
<locationhearder>:="http://www."|"http://"|"file://"|"ftp://ftp."
<path>:=<directory+><word>''.''<word>
<directory>:=<word>
<participantcount>:=<digit+>|"n"
<return>:="operationresult"|"information"|"agentaddress"
<roles>:=<role><role>|<roles><role>
<messagepatterns>:=<acl><messagepattern+>
<role>:=<roleproperties><variables?><actions?><phases>
<roledescriptors>:=<roledescriptors><roleattributes><rolekeywords?>
<roledescriptors>:=<identifier><name>
<roleattributes>:=<cardinality><concurrentparticipants?>
<concurrentparticipantset>:=<identifier+>
<rolekeywords>:=<rolekeyword+>
<rolekeyword>:=<word>
```

```
<name>:=<word>
<cardinality>:=<digit+>|"n"
<variables>:=<variable+>
<variable>:=<identifier><type>
<type>:="number"|"string"|"char"|"boolean"|"date"|
               "collection"|"null"|"any"|"map"
<identifier>:=<letter+> "id" <digit+>
<letter>:="a"|"b"|"c"|..."z"
<digit>:="0"|"1"|"2"|...|"9"
<word>:=<letter+>
<space>:=""
<actions>:=<action+>
<phases>:=<phase+>
<phase>:=<identifier><actions>
<action>:=<category><description?><signature?><events>
<description>:=(<word><space?>)*
<category>:="append"|"custom"|"remove"|"send"|"set"|"update"
<signature>:=<arguments>|<messages>
<arguments>:=(<argset>|<argdesc>)+
<argset>:=<settype>(<argset>|<argdesc>)+
<argdesc>:=<identifier><type><direction>
<direction>:="in"|"out"|"inout"
<messages>(<message>|<messageset>)+
<message>:=<identifier>
<messageset>:=<settype>(<messageset>|<message>)+
<settype>:="and"|"or"|"xor"
<events>:=(<event>|<eventref>|<eventset>)+
<eventset>:=<settype>(<event>|<eventref>|<eventset>)+
<event>:=<identifier?><eventtype><object>
<eventtype>:="change"|"custom"|"emission"|"endphase"|
           "endprotocol"|"messagecontent"|"reception"|"variablecontent"
<object>:=<message>|<variableid>
<variableid>:=<identifier>
<eventref>:=<identifier>
<messagepattern>:=<identifier><performative><content>
<performative>:=<fipaperformative>|<kqmlperformative>
<kqmlperformative>:="ask-all"|"ask-one"|"ask-if"|"stream-all"|...
<acl>:="fipa"|"kqml"
<content>:=<type><pattern?>
<pattern>:=<word*><space>
```

Plan Generation and Plan Execution in Agent Programming

M. Birna van Riemsdijk and Mehdi Dastani

Institute of Information and Computing Sciences
Utrecht University
The Netherlands
{birna, mehdi}@cs.uu.nl

Abstract. *This paper presents two approaches for generating and executing the plans of cognitive agents. They can be used to define the semantics of programming languages for cognitive agents. The first approach generates plans before executing them while the second approach interleaves the generation and execution of plans. Both approaches are presented formally and their relation is investigated.*

1 Introduction

Various programming languages have been proposed to implement cognitive agents [14,2,8,6,9,12,5,7,11]. These languages provide data structures to represent the agent's mental attitudes such as beliefs, goals and plans. Beliefs describe the state of the world the agent is in, goals describe the state the agent wants to reach and plans are the means to achieve these goals.

Most of these programming languages can be viewed as inspired in some way by the Procedural Reasoning System (PRS) [6]. This system was proposed as an alternative to the traditional planning systems [13], in which plans to get from a certain state to a goal state are constructed by reasoning about the results of primitive actions. PRS and most of today's cognitive agent programming languages, by contrast, use a library of pre-specified plans.[1] The goals for the achievement of which these plans can be selected, are part of the plan specification. Further, plans might not consist of primitive actions only, but they can also contain subgoals. If a subgoal is encountered during the execution of a plan, a plan for achieving this subgoal should be selected from the plan library, after which it can be executed. An agent can for example have the plan to take the bus into town, to achieve the subgoal of having bought a birthday cake, and then to eat the cake.[2] This subgoal of buying a birthday cake will have to be fulfilled by selecting and executing in turn an appropriate plan of for example which shops to go to, paying for the cake, etc., before the agent can execute

[1] The language ConGolog [7], in which the agent reasons about the result of the execution of its actions, is an exception.

[2] Assuming that both taking the bus into town and eating cake are primitive actions that can be executed directly.

the action of eating the cake. Plans containing subgoals are called *partial* plans, while plans containing only primitive actions are called *total*.

An important advantage of PRS and similar systems over traditional planning systems is that they do not require search through potentially large search spaces. A disadvantage of PRS-like systems has to do with the fact that most of these systems allow for multiple plans to be executed concurrently, i.e., the agent may pursue multiple goals simultaneously. These plans can conflict, as they, for example, can require the same resources. In PRS-like systems, in which plans for subgoals are selected during execution of the plan, it is difficult to predict whether plans will conflict. If a plan containing subgoals is selected, it is not yet known how the subgoals of this plan will be achieved. It is therefore difficult to assess whether this plan will conflict with other plans of the agent.

One way to approach this issue, is to use a representation of plans that contains information that can be used to detect possible conflicts among plans, as proposed by Thangarajah et al. [16,15]. Once these conflicts are detected, plans can be scheduled in such a way that conflicts do not occur during execution of the plans.

In this paper, we take a slightly different approach. That is, in order to be able to compare an approach in which information about conflicting plans is taken into account with an approach of *plan execution* in the PRS style, we take an operational approach to the former, which we call *plan generation*. The idea of plan generation is to use pre-specified partial plans to generate total plans offline, i.e., before the plans are executed. Since conflicts among plans generally depend on the primitive actions within the plans, the generation of total plans provides for the possibility to check whether plans are conflicting. We assume that a specification of conflicts among plans is given, e.g., in a way comparable with the work of Thangarajah et al.

In order to compare plan generation with plan execution, we first introduce a framework for plan generation (Section 2). This framework defines how non-conflicting sets of plans can be generated on the basis of a plan library (i.e., rules for selecting plans to achieve (sub)goals), a set of top-level goals, and a set of initial partial plans. These definitions are inspired by default logic. In default logic, various so-called extensions, which consist of consistent sets of first-order formulas, can be derived on the basis of possibly conflicting default rules, and an initial set of facts. The fact that default rules might conflict, gives rise to the possibility of deriving multiple extensions on the basis of a single default theory. We adapt the notion of extension as used in default logic, to the context of conflicting plans. An extension then consists of a set of non-conflicting plans. The idea of adapting the notion of extension as used in default logic to the context of plans, is inspired by the BOID framework [2]. It was however not worked out in detail in the cited paper.

The language we use as an example of a PRS style framework, is a simplified version of the cognitive agent programming language 3APL [8,3], and is presented in Section 3. We assume that a specification of conflicts among plans is given. Ways of specifying conflicts have been investigated in the literature (see, e.g., [16]), and further research along these lines is beyond the scope of this paper. We show in Section 4 that, for any total plan in an extension of a so-called plan generation agent, there is a corresponding initial plan in the execution setting, which has the same semantics. If one would assume that in an offline plan generation context, a single extension is chosen for execution, one could say that the behavior of a plan generation agent is "included" in the behavior of a plan execution agent. This is intuitive, since the incorporation of a notion of conflict among plans *restricts* the set of plans which can be executed concurrently.

2 Plan Generation

In this section, we present a framework for plan generation that is based on [2]. In that paper, a non-standard approach to planning is taken, in which rules are used to specify which plan can be adopted for a certain goal. This is in contrast with planning from first principles, in which action specifications are taken as the basis, and a sequence of actions is sought that realizes a certain goal state according to the action specifications, given an initial situation. In [2] and in the current paper, it is the job of the agent *programmer* to specify which (composed) plan (or plan recipe) is appropriate for which goal.

Throughout this paper, we assume a language of propositional logic $\mathcal{L}$ with negation and conjunction, with typical element ϕ. The symbol $\models$ will be used to denote the standard entailment relation for $\mathcal{L}$.

Below, we define the language of plans. A plan is a sequence of basic actions and $achieve(\phi)$ statements, the latter representing that the goal ϕ is to be achieved. In correspondence with the semantics of 3APL, basic actions change an agents beliefs when executed. This will be defined formally in Section 3. One could add a test statement and non-deterministic choice, but we leave these out for reasons of simplicity. A total plan is a plan containing only basic actions.

Definition 1 *(plans)* Let $\mathsf{BasicAction}$ with typical element a be a set of basic actions and let $\phi \in \mathcal{L}$. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid achieve(\phi) \mid \pi_1; \pi_2$$

The set of total plans $\mathsf{TotalPlan}$ is the subset of Plan containing no $achieve(\phi)$ statements. We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

Before we define the notion of an agent, we define the rules that represent which plan can be adopted to achieve a certain goal. These plan generation rules have a propositional formula as the head, representing the goal, and a plan as the body. In principle, plan generation rules can be extended to include a belief condition in the head, indicating that the plan in the body can be adopted if the agent has a certain goal *and* a certain belief. The belief condition could then be viewed as the precondition of the plan. For reasons of simplicity, we however define rules as having only a condition on goals.

Definition 2 *(plan generation rule)* The set of plan generation rules $\mathcal{R}_{\mathsf{PG}}$ is defined as follows: $\mathcal{R}_{\mathsf{PG}} = \{\phi \Rightarrow \pi \mid \phi \in \mathcal{L}, \pi \in \mathsf{Plan}\}$.

A plan generation agent is a tuple consisting of a belief base, a goal base, a plan base and a rule base. The belief base and goal base are consistent. The rule base consists of a set of plan generation rules and may not contain multiple rules for the same goal. This prevents that multiple plans for the same goal can be adopted, which could be considered undesirable. The plans base contains the initial set of plans of the agent.

Definition 3 *(plan generation agent)* A plan generation agent[3], typically denoted by $\mathcal{A}$, is a tuple $\langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ where $\sigma \subseteq \mathcal{L}$ is the belief base, $\gamma \subseteq \mathcal{L}$ is the goal base, $\mathit{\Pi} \subseteq \mathsf{Plan}$ is the plan base and $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$ is a set of rules. Further, $\sigma \not\models \bot$ and $\gamma \not\models \bot$ and all sets $\sigma, \gamma, \mathit{\Pi}$ and PG are finite. Finally, PG does not contain multiple rules with an equivalent head, i.e., if $\phi \Rightarrow \pi \in \mathsf{PG}$, there is not a rule $\phi' \Rightarrow \pi' \in \mathsf{PG}$ such that $\phi \equiv \phi'$.

When generating plans, we want to take into account conflicts, for example with respect to resources, that may arise among plans. For this, we assume a notion of coherency of plans. A plan π being coherent with a set of plans $\mathit{\Pi}$ will be denoted by $\mathit{coherent}(\pi, \mathit{\Pi})$. We assume that once a (partial) plan is incoherent with a set of plans, this plan cannot become coherent again by refining the plan, i.e., by replacing a subgoal with a more concrete plan.

We are now in a position to define how a coherent set of plans is generated on the basis of an agent $\langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$. A natural way in which to define this plan generation process, is an approach inspired by default logic. In default logic, consistent sets of formulas or *extensions* are generated on the basis of a possibly conflicting set of default rules, and a set of formulas representing factual world knowledge. Here, we generate sets of coherent plans on the basis of an initial set of plans $\mathit{\Pi}$, a goal base γ, and a set of plan generation rules PG.

The idea is that we take the plan base $\mathit{\Pi}$ of the agent, which may contain partial plans, as the starting point. These partial plans in $\mathit{\Pi}$ are refined by means of applying plan generation rules from PG. If $\pi_1; \mathit{achieve}(\phi); \pi_2$ is a plan in $\mathit{\Pi}$ and $\phi \Rightarrow \pi$ is a rule in PG, then this rule can be applied, yielding the plan $\pi_1; \pi; \pi_2$. This process can continue, until total plans are obtained. Further, a plan generation rule $\phi \Rightarrow \pi$ can be applied if ϕ follows from the goal base γ. In that case, a new plan π is added to the existing set of plans, which can in turn be refined through rule applications.

The plans that are generated in this way should however be mutually coherent. A plan can thus only be added to the existing set of plans through refinement or plan addition, if this plan is coherent with already existing ones. Different choices of which plan to refine or to add may thus have different outcomes in terms of the resulting set of coherent plans: the addition of a plan may prevent the addition of other plans that are incoherent with this plan.

[3] In this section we take the term "agent" to mean "plan generation agent".

Differing from [2], we define the notion of an extension in the context of plans through the notion of a *process*. This is based on the concept of a process as used in [1] to define extensions in the context of default logic. A process is a sequence of sets of plans, such that each consecutive set is obtained from the previous by applying a plan generation rule. A process can formally be defined in terms of a transition system which is a set of transition rules that indicate the transitions between consecutive sets of plans.

Given a set of plans E_i and an agent $\langle \sigma, \gamma, \Pi, \mathsf{PG} \rangle$, a rule $\phi \Rightarrow \pi \in \mathsf{PG}$ can be applied if ϕ follows from γ. The plan π is then added to E_i, that is, if $\pi \notin E_i$ and $coherent(\pi, E_i)$. This rule can also be applied if there is a plan of the form $\pi_1; achieve(\phi); \pi_2$ in E_i.[4] In that case, the plan $\pi_1; \pi; \pi_2$ is added to E_i, again only if the plan is not already in E_i and it is coherent with E_i. One could also remove the original plan $\pi_1; achieve(\phi); \pi_2$ from E_i, but addition of the refined plan is more in line with the definition of processes and extensions in default logic. It would be more useful if a plan of the form $\pi_1; achieve(\phi); \pi_2$ could be refined by a rule $\phi' \Rightarrow \pi$ if $\phi \equiv \phi'$, but we omit this extra clause to simplify our definitions. The first element of a process of an agent is the plan base Π of the agent.

Definition 4 *(process)* Let $\mathcal{A} = \langle \sigma, \gamma, \Pi, \mathsf{PG} \rangle$ be an agent. A sequence of sets $E_0, \ldots, E_n$ with $E_i \subseteq \mathsf{Plan}$ is a process of $\mathcal{A}$ iff $E_0 = \Pi$ and it holds for all E_i with $0 \leq i \leq n-1$ that $E_i \to E_{i+1}$ is a transition that can be derived in the transition system below. Let $\phi \Rightarrow \pi \in \mathsf{PG}$ be a plan generation rule. The transition rule for *plan addition* is then defined as follows:

$$\frac{\gamma \models \phi \qquad \pi \notin E \qquad coherent(\pi, E)}{E \to E'}$$

where $E' = E \cup \{\pi\}$. The transition rule for *plan refinement* is defined as follows:

$$\frac{\begin{array}{c}\pi_1; achieve(\phi); \pi_2 \in E \qquad \pi_1; \pi; \pi_2 \notin E \\ coherent(\pi_1; \pi; \pi_2, E)\end{array}}{E \to E'}$$

where $\pi_1, \pi_2 \in \mathsf{Plan}$ and $E' = E \cup \{\pi_1; \pi; \pi_2\}$.

We assume that the plan generation rules of an agent are such that no infinite processes can be constructed on the basis of the corresponding transition system.

The notion of an extension is defined in terms of the notion of a closed process. A process is closed iff no rules are applicable to the last element of the process. This is formalized in the definitions below. Note that not all processes are closed. A closed process can be viewed as a process that has terminated, i.e., there are no transitions possible from the last element in the process. It is however the case that we assume that any process can become a closed process.

[4] Note that, for example, a plan $achieve(\phi)$ is also of this form, as π_1 and π_2 can be the empty plan ϵ (see Definition 1).

Definition 5 *(applicability)* A plan generation rule $\phi \Rightarrow \pi$ is applicable to a set $E \subseteq \mathsf{Plan}$ iff a transition $E \rightarrow E'$ can be derived in the transition system above on the basis of this rule.

Definition 6 *(closed process)* A process $E_0, \ldots, E_n$ of an agent $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ is closed iff there is not a plan generation rule $\delta \in \mathsf{PG}$ such that δ is applicable to E_n.

Definition 7 *(extension)* A set $E \subseteq \mathsf{Plan}$ is an extension of $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ iff there is a closed process $E_0, \ldots, E_n$ of $\mathcal{A}$ such that $E = E_n$.

The execution of a plan generation agent is as follows. An extension of the agent is generated. This extension is a coherent set of partial and total plans. The total plans can then be executed according to the semantics of execution of basic actions as will be provided in Section 3.

3 Plan Execution

In this section, we present a variant of the agent programming language 3APL, which suits our purpose of comparing the language with the plan generation framework of the previous section. An important component of 3APL agents that we need in this paper, is the so-called plan revision rules which have a plan as the head and as the body. During execution of a plan, a plan revision rule can be used to replace a prefix of the plan, which is identical to the head of the rule, by the plan in the body. If the agent for example executes a plan $a; b; c$ and has a plan revision rule $a; b \Rightarrow d$, it can apply this rule, yielding the plan $d; c$.

Here we do not need the general plan revision rules that can have a composed plan as the head. We only need rules with statements of the form $\mathit{achieve}(\phi)$ as the head and a plan as the body.

Definition 8 *(plan revision rule)* The set of plan revision rules $\mathcal{R}_{\mathsf{PR}}$ is defined as follows: $\mathcal{R}_{\mathsf{PR}} = \{\mathit{achieve}(\phi) \Rightarrow \pi \mid \phi \in \mathcal{L}, \pi \in \mathsf{Plan}\}$.

An agent in this context is similar to the plan generation agent of Definition 3, with a rule base consisting of a set of plan revision rules. The rule base may not contain multiple rules for the same $\mathit{achieve}(\phi)$ statement. We also introduce a function $\mathcal{T}$ that takes a belief base σ and a basic action a and yields the belief base resulting from executing a in σ. This function is needed in order to define the semantics of plan execution. We use $\mathit{\Sigma} = \wp(\mathcal{L})$ to denote the set of belief bases.

Definition 9 *(plan execution agent)* Let $\mathcal{T} : (\mathsf{BasicAction} \times \mathit{\Sigma}) \rightarrow \mathit{\Sigma}$ be a function specifying the belief update resulting from the execution of basic actions. A plan execution agent, typically denoted by $\mathcal{A}'$, is a tuple $\langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PR}, \mathcal{T} \rangle$, where $\sigma \subseteq \mathcal{L}$ is the belief base, $\gamma \subseteq \mathcal{L}$ is the goal base, $\mathit{\Pi} \subseteq \mathsf{Plan}$ is the plan base and $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$ is a set of plan revision rules. Further, $\sigma \not\models \bot$ and $\gamma \not\models \bot$ and all sets σ, γ, Π and PR are finite. The rule base PR does not contain multiple rules with an equivalent head, i.e., if $achieve(\phi) \Rightarrow \pi \in \mathsf{PR}$, there is not a rule $achieve(\phi') \Rightarrow \pi' \in \mathsf{PR}$ such that $\phi \equiv \phi'$.

We can now move on to defining the semantics of plan execution. As it will become clear, we only need the semantics of individual plans for the relation between plan generation and plan execution that we will establish in Section 4. The semantics of executing a plan base containing a set of plans can be defined by interleaving the semantics of individual plans (see [8]).

The semantics of a programming language can be defined as a function taking a statement (plan) and a state (beliefbase), and yielding the set of states resulting from executing the initial statement in the initial state. In this way, a statement can be viewed as a transformation function on states. There are various ways of defining a semantic function and in this paper we are concerned with the so-called *operational* semantics [4].

The operational semantics of a language is usually defined using transition systems [10]. A transition system for a programming language consists of a set of axioms and derivation rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. A configuration is here a tuple $\langle \pi, \sigma \rangle$, consisting of a plan π and a belief base σ. Below, we give the transition system $\mathsf{Trans}_{\mathcal{A}'}$ that defines the semantics of plan execution. This transition system is specific to agent $\mathcal{A}'$.

There are two kinds of transitions, i.e., transitions describing the execution of basic actions and those describing the application of a plan revision rule. The transitions are labelled to denote the kind of transition. A basic action at the head of a plan can be executed in a configuration if the function $\mathcal{T}$ is defined for this action and the belief base in the configuration. The execution results in a change of belief base as specified through $\mathcal{T}$ and the action is removed from the plan.

Definition 10 *($\mathsf{Trans}_{\mathcal{A}'}$)* Let $\mathcal{A}'$ be a plan execution agent with a set of plan revision rules PR and a belief update function $\mathcal{T}$. The transition system $\mathsf{Trans}_{\mathcal{A}'}$, consisting of a transition rule for action execution and one for rule application, is defined as follows. Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle a; \pi, \sigma \rangle \rightarrow_{exec} \langle \pi, \sigma' \rangle}$$

Let $achieve(\phi) \Rightarrow \pi \in \mathsf{PR}$.

$$\langle achieve(\phi); \pi', \sigma \rangle \rightarrow_{apply} \langle \pi; \pi', \sigma \rangle$$

Note that the goal base is not used in this semantics. Based on this transition system, we define the operational semantic function below. This function takes an initial plan and belief base. It yields the belief base resulting from executing the plan on the initial belief base, as specified through the transition system.

Definition 11 *(operational semantics)* Let $x_i \in \{exec, apply\}$ for $1 \leq i \leq n$. The operational semantic function $\mathcal{O}^{\mathcal{A}'} : \mathsf{Plan} \to (\Sigma \to \Sigma)$ is a partial function that is defined as follows.

$$\mathcal{O}^{\mathcal{A}'}(\pi)(\sigma) = \begin{cases} \sigma_n & \text{if } \langle \pi, \sigma \rangle \to_{x_1} \ldots \to_{x_n} \langle \epsilon, \sigma_n \rangle \text{ is a finite sequence of} \\ & \text{transitions in } \mathsf{Trans}_{\mathcal{A}'} \\ \text{undefined otherwise} & \end{cases}$$

The result of executing a plan is a single belief base, as plan execution as defined in this paper is deterministic: in any configuration, there is only one possible next configuration (or none). See for example [17] for a specification of the semantics of plan execution in case of non-determinism.

4 Relation between Plan Generation and Plan Execution

In this section, we will investigate how these two are related. In order to do this, we first define a function f, which transforms plan generation rules into plan revision rules of a similar form.

Definition 12 *(plan generation rules to plan revision rules)* The function $f : \wp(\mathcal{R}_{\mathsf{PG}}) \to \wp(\mathcal{R}_{\mathsf{PR}})$, transforming plan generation rules into plan revision rules, is defined as follows: $f(\mathsf{PG}) = \{achieve(\phi) \Rightarrow \pi \mid \phi \Rightarrow \pi \in \mathsf{PG}\}$.

The theorem we prove, relates the operational semantics of the total plans of an extension of a plan generation agent, to the plans in the initial plan base of a corresponding plan execution agent. It says that for any total plan α in the extension, there is a plan π in the plan base of the plan execution agent, such that the operational semantics of α and π are equivalent. The plan α is a plan from the plan generation agent and we have not defined an operational semantics in this context. We however take for the operational semantics of α the operational semantics for plans as defined in the context of plan execution agents. Note though that, for the semantics of α, only the *exec* transition of the transition system on which the operational semantics is based, is relevant.[5]

The intuition as to why this relation would hold, is the following. The generation of a total plan α from a partial plan π under a set of plan generation rules PG, corresponds with the execution of π, under a set of plan revision rules $f(\mathsf{PG})$. The plan revision rules applied during execution of π have a plan generation counterpart that is applied during generation of α. Further, the basic actions that are executed during the execution of π, are precisely the basic actions of α (in the same order). Because of this, the operational semantics of α and π are equivalent, as the execution of basic actions completely determines the changes to the initial belief base, and therefore the belief base at the end of the execution.

[5] We could have defined a new transition system for total plans, only containing the *exec* transition of the system of Definition 10, and a corresponding operational semantics. This is straightforward, so we omit this.

If $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ is a plan generation agent, the rule base of the corresponding plan execution agent $\mathcal{A}'$ should thus be $f(\mathsf{PG})$. For the belief base and goal base of $\mathcal{A}'$, we take σ and γ, respectively. As for the plan base of $\mathcal{A}'$, we cannot just take $\mathit{\Pi}$, for the following reason. A total plan α in an extension of $\mathcal{A}$ can be generated either from a partial plan π that was already in $\mathit{\Pi}$, or from a plan π that has been added by applying a plan generation rule to the goal base (through a plan addition transition in the process). If the latter is the case, we have to make sure that π is in the plan base of $\mathcal{A}'$, as this is the plan of which the semantics is equivalent with α. We thus define that the plan base of $\mathcal{A}'$ is $\mathit{\Pi} \cup \{\pi \mid \mathit{achieve}(\phi) \Rightarrow \pi \in f(\mathsf{PG}), \gamma \models \phi\}$. We now have the following theorem.

Theorem 1 Let $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ be an agent and let E be an extension of $\mathcal{A}$. Let $\mathcal{A}' = \langle \sigma, \gamma, \mathit{\Pi}', f(\mathsf{PG}), \mathcal{T} \rangle$ where $\mathit{\Pi}' = \mathit{\Pi} \cup \{\pi \mid \mathit{achieve}(\phi) \Rightarrow \pi \in f(\mathsf{PG}), \gamma \models \phi\}$ and let $\alpha \in \mathsf{TotalPlan}$. We then have the following.

$$\forall \alpha \in E : \exists \pi \in \mathit{\Pi}' : \mathcal{O}^{\mathcal{A}'}(\alpha)(\sigma) = \mathcal{O}^{\mathcal{A}'}(\pi)(\sigma)$$

In order to prove this theorem, we need a number of auxiliary definitions and lemmas. The first is the notion of an extended process. The idea is, that we want to derive from a given process p and a given total plan α in the extension corresponding with p, those steps in p that lead from some initial partial plan π to α. For this, we give each plan in the plan base of the agent a unique number. Then, we associate with each step in the process the number of the plan that is being refined. If a plan is added through a plan addition transition, we give this new plan a unique number and associate this number with the transition step.

The elements of the sets of an extended process are thus pairs from $\mathsf{Plan} \times \mathbb{N}$. A pair $(\pi, i) \in (\mathsf{Plan} \times \mathbb{N})$ will be denoted by π^i. We use the notion of a natural number i being fresh in E to indicate uniqueness of i in E: i is fresh in E if there is not a plan π^i in E.[6] Further, a rule $\phi \Rightarrow \pi$ can only be applied to refine a plan $\pi_1; \mathit{achieve}(\phi); \pi_2$, if $\mathit{achieve}(\phi)$ is the leftmost *achieve* statement of the plan, i.e., if π_1 is a total plan. This corresponds more closely with the application of plan revision rules in plan execution, as during execution always the first (or leftmost) *achieve* statement of a plan is rewritten.

Definition 13 *(extended process)* Let $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ be a plan generation agent and let $I(\mathit{\Pi})$ be $\mathit{\Pi}$ where each plan in $\mathit{\Pi}$ is assigned a unique natural number. A sequence of sets, alternated with natural numbers, $E_0, i_1, E_1, \ldots, i_n, E_n$ with $E_i \subseteq \mathsf{Plan}$ and $i_j \in \mathbb{N}$ with $1 \leq j \leq n$ is an extended process of $\mathcal{A}$ iff $E_0 = I(\mathit{\Pi})$ and it holds for all triples E_k, i, E_{k+1} in this sequence that $E_k \rightarrow_i E_{k+1}$ is a transition that can be derived in the transition system below.

Let $\phi \Rightarrow \pi \in \mathsf{PG}$ be a plan generation rule. The transition rule for *plan addition* is then defined as follows:

$$\frac{\gamma \models \phi \qquad \pi \notin E \qquad \mathit{coherent}(\pi, E)}{E \rightarrow_i E'}$$

[6] We refer to the pairs π^i as plans and we will from now on take the set Plan as including both ordinary plans π and pairs π^i.

where $E' = E \cup \{\pi^i\}$ with i fresh in E. The transition rule for *plan refinement* is defined as follows:

$$\frac{(\alpha_1; achieve(\phi); \pi_2)^i \in E \qquad (\alpha_1; \pi; \pi_2)^i \notin E \qquad coherent(\alpha_1; \pi; \pi_2, E)}{E \rightarrow_i E'}$$

where $\alpha_1 \in \mathsf{TotalPlan}$, $\pi_2 \in \mathsf{Plan}$ and $E' = E \cup \{(\alpha_1; \pi; \pi_2)^i\}$.

The notion of a closed process (Definition 6) as defined for processes in Definition 4, is applied analogously to extended processes.

We will prove theorem 1 using the notion of an extended process. Theorem 1 is however defined in terms of an extension, which is defined in terms of ordinary processes, rather than extended processes. We thus have to show that extended processes and processes are equivalent in some sense. We show that for any closed process there is a closed extended process that has the same final set of plans, with respect to the total plans in this set. We only provide a brief sketch of the proof.

Lemma 1 *(process equivalence)* Let $\mathcal{A}$ be a plan generation agent and let $t : \wp(\mathsf{Plan}) \rightarrow \wp(\mathsf{TotalPlan})$ be a function yielding the total plans of a set of plans. The following then holds: there is a closed process $E_0, \ldots, E_n$ of $\mathcal{A}$, iff there is a closed extended process $E'_0, i_1, E'_1, \ldots, i_n, E'_n$ of $\mathcal{A}$ such that $t(E_n) = t(E'_n)$ (modulo superscripts of plans).

Sketch of proof: ($\Leftarrow$) If a transition $E \rightarrow_i E'$ can be derived in the transition system of Definition 13, then a transition $E \rightarrow E'$ can be derived in the system of Definition 4 (modulo superscripts). ($\Rightarrow$) This is proven by viewing the plan generation rules as the production rules of a grammar and the total plans that can be generated by these rules as the language of this grammar. The formulas ϕ and the statements $achieve(\phi)$ are considered the non-terminals of the grammar and the set of basic actions $\mathsf{BasicAction}$ the terminals. The plans of the first element of an (extended) process can be viewed as the start symbols of the grammar, together with those plans that are added through the transition rule for plan addition.

It is the case that for any derivation of a string (or total plan) in the grammar, an equivalent leftmost derivation, in which at each derivation step the leftmost non-terminal is rewritten, can be constructed. Derivations in an extended process correspond with leftmost derivations, from which the desired result can be concluded. $\square$

Given a closed extended process p with E_n as its final element, and a total plan $\alpha^i \in E_n$, we are interested in those steps of p that lead to the derivation of α^i. In other words, we are interested in those steps that are labelled with i. For this, we define the notion of an i-process of an extended process. This consists of a sequence of pairs of sets of plans, where each pair corresponds with a derivation step that is labelled with i, in the original extended process.

Given the i-process p_i of an extended process p, we define the notion of the i-derivation of p_i. The i-derivation of p_i is the sequence of singleton sets of plans,[7] that is yielded by subtracting for each pair (E, E') occurring in p_i, the set E from the set E'. An i-derivation is thus a sequence $\pi_1^i, \pi_2^i, \ldots, \pi_m^i$,[8] in which each plan is labelled with i. The sequence can be viewed as the derivation of the plan π_m^i from the initial plan π_1^i, as each step from π_j^i to π_{j+1}^i in this sequence corresponds with the application of a plan generation rule to π_j^i, yielding π_{j+1}^i.

Definition 14 *(i-derivation)* Let $\mathcal{A} = \langle \sigma, \gamma, \Pi, \mathsf{PG} \rangle$ be a plan generation agent and let $p = E_0, i_1, E_1, \ldots, i_n, E_n$ be a closed extended process of $\mathcal{A}$. The i-process p_i of p is then defined as a sequence of pairs $(E'_0, E'_1), \ldots, (E'_{m-1}, E'_m)$ such that the following holds: (E, E') occurs in p_i iff E, i, E' occurs in p and for any two consecutive pairs $(E_j, E_{j+1}), (E_{j+2}, E_{j+3})$ occurring in p_i it should hold that $E_{j+1} \subseteq E_{j+2}$.

Let $p_i = (E_0, E_1), \ldots, (E_{m-1}, E_m)$ be the i-process of a closed extended process p. The i-derivation of p_i is then defined as follows: $(E_1 \setminus E_0), \ldots, (E_m \setminus E_{m-1})$.

We want to associate the semantics of a total plan α in some extension of a plan generation agent, with the semantics of a corresponding plan π in the initial plan base of a plan execution agent. We do this by showing that the basic actions executed during the execution of π, correspond exactly with the basic actions of α. For this, we define a variant of the transition system of Definition 10, in which the configurations are extended with a third element. This element, which is a total plan, represents the basic actions that have been executed so far in the execution. Further, we define the execution of a sequence of basic actions in one transition step. This is convenient when proving lemma 2.

Definition 15 *($\mathsf{Trans}'_{\mathcal{A}'}$)* Let $\mathcal{A}'$ be a plan execution agent with a set of plan revision rules PR and a belief update function $\mathcal{T}$. The transition system $\mathsf{Trans}'_{\mathcal{A}'}$, consisting of a transition rule for action execution and one for rule application, is defined as follows.

Let $\alpha \in \mathsf{TotalPlan}$ be a sequence of basic actions and let $\mathcal{T}' : (\mathsf{TotalPlan} \times \Sigma) \to \Sigma$ be the lifting of $\mathcal{T}$ to sequences of actions, i.e., $\mathcal{T}'(a;\alpha)(\sigma) = \mathcal{T}'(\alpha)(\mathcal{T}(a)(\sigma))$. Further, let $\alpha' \in \mathsf{TotalPlan}$ be a sequence of basic actions, representing the actions that have already been executed.

$$\frac{\mathcal{T}'(\alpha, \sigma) = \sigma'}{\langle \alpha; \pi, \sigma, \alpha' \rangle \rightarrow_{exec} \langle \pi, \sigma', \alpha'; \alpha \rangle}$$

Let $achieve(\phi) \Rightarrow \pi \in \mathsf{PR}$.

$$\langle achieve(\phi); \pi', \sigma, \alpha \rangle \rightarrow_{apply} \langle \pi; \pi', \sigma, \alpha \rangle$$

[7] It is a sequence of *singleton* sets, as each pair in an i-process corresponds with a derivation step in the original process. In a derivation step from E to E', exactly one plan is added to E.

[8] We omit curly brackets.

It is easy to see that an operational semantics $\mathcal{O}'$ can be defined[9] on the basis of this transition system that is equivalent with the operational semantics of Definition 11, i.e., such that $\mathcal{O}'(\pi)(\sigma) = \mathcal{O}(\pi)(\sigma)$ for any plan π and belief base σ. The initial configuration of any transition sequence in $\mathsf{Trans}'_{\mathcal{A}'}$ should be of the form $\langle \pi, \sigma, \epsilon \rangle$, as the third element represents the sequence of actions that have been executed, which are none in the initial configuration.

In the proof of lemma 2, we use the notion of a maximum prefix of a plan.

Definition 16 *(maximum prefix)* Let $\alpha \in \mathsf{TotalPlan}$ and let $\pi \in \mathsf{Plan}$. We then say that α is a maximum prefix of π iff $\alpha = \pi$ or $\pi = \alpha; \mathit{achieve}(\phi); \pi'$. Note that π' can be ϵ.

Lemma 2 says the following. Let α^i be a total plan in a closed extended process of a plan generation agent, and let π_1^i be the first plan of the i-derivation of α^i. It is then the case that the actions executed during the execution of π_1 (given an appropriate set of plan revision rules), are exactly the actions of α (in the same order).

Lemma 2 Let $\mathcal{A} = \langle \sigma, \gamma, \mathit{\Pi}, \mathsf{PG} \rangle$ be a plan generation agent and let $p = E_0, i_1, E_1, \ldots, i_n, E_n$ be a closed extended process of $\mathcal{A}$. Let $\alpha^i \in E_n$ where $\alpha \in \mathsf{TotalPlan}$. Further, let $\pi_1^i, \ldots, \alpha^i$ be the i-derivation of the i-process p_i of p. Let $\mathcal{A}' = \langle \sigma, \gamma, \mathit{\Pi}', f(\mathsf{PG}), \mathcal{T} \rangle$ be a plan execution agent where $\mathit{\Pi}' = \mathit{\Pi} \cup \{\pi \mid \mathit{achieve}(\phi) \Rightarrow \pi \in f(\mathsf{PG}), \gamma \models \phi\}$. Further, let $\mathcal{T}'(\alpha)(\sigma)$ be defined and let $x_i \in \{\mathit{exec}, \mathit{apply}\}$ for $1 \leq i \leq m-1$. The following then holds.

A transition sequence of the form

$$\langle \pi_1, \sigma, \epsilon \rangle \to_{x_1} \ldots \to_{x_{m-1}} \langle \epsilon, \sigma_m, \alpha \rangle \quad \text{can be derived in } \mathsf{Trans}'_{\mathcal{A}'}. \tag{4.1}$$

Sketch of proof: We say that a plan π^i corresponds with a configuration $\langle \pi', \sigma, \alpha \rangle$ iff $\pi = \alpha; \pi'$. Let π_k^i and π_{k+1}^i be two consecutive plans in the i-derivation of p_i, where π_k^i is of the form $\alpha_2; \mathit{achieve}(\phi_2); \pi_2$ and π_{k+1}^i is of the form $\alpha_2; \pi; \pi_2$. This corresponds with the application of plan generation rule $\phi_2 \Rightarrow \pi$. Let π be of the form $\alpha_3; \mathit{achieve}(\phi_3); \pi_3$. We then have that the following transition sequence can be derived in $\mathsf{Trans}'_{\mathcal{A}'}$.

$$\langle \mathit{achieve}(\phi_2); \pi_2, \sigma, \alpha_2 \rangle \to_{\mathit{apply}} \langle \alpha_3; \mathit{achieve}(\phi_3); \pi_3; \pi_2, \sigma, \alpha_2 \rangle \to_{\mathit{exec}} \langle \mathit{achieve}(\phi_3); \pi_3; \pi_2, \sigma', \alpha_2; \alpha_3 \rangle \tag{4.2}$$

This pair of transitions is correspondence and maximum prefix preserving. If π_1 (transition sequence (4.1)) is of the form $\alpha_1; \mathit{achieve}(\phi_1); \pi$, we can derive a transition in which α_1 is executed. This yields a configuration of the form

[9] We omit superscript $\mathcal{A}'$.

$\langle achieve(\phi_1); \pi, \sigma', \alpha_1\rangle$, which corresponds with π_1^i and for which it holds that α_1 is a maximum prefix of π_1. From this configuration, a sequence of *apply* and *exec* transitions can be derived, given that we have (4.2) for every pair π_k^i and π_{k+1}^i occurring in the i-derivation. From the fact that this sequence of transitions is correspondence and maximum prefix preserving, we can conclude that the final configuration $\langle \pi_m, \sigma_m, \alpha_m\rangle$ of the sequence must be of the form $\langle \epsilon, \sigma_m, \alpha\rangle$ (observe that α_i is the final plan of the i-derivation, which should correspond with $\langle \pi_m, \sigma_m, \alpha_m\rangle$). □

We are now in a position to prove theorem 1.

Proof of theorem 1 (sketch): We do not repeat the premisses of the theorem. Let $\alpha \in E$ be a total plan in E. By lemma 1, we then have that there is a closed extended process with a final set E_n such that $\alpha^i \in E_n$ for some natural number i. Let $\pi_1^i, \ldots, \alpha^i$ be the corresponding i-derivation. The plan π_1 was either added in the process through a plan addition transition, or it was already in Π. From this we can conclude that $\pi_1 \in \Pi'$.

If $\mathcal{T}'(\alpha)(\sigma)$ is defined, we have by lemma 2 that a transition sequence of the form $\langle \pi_1, \sigma, \epsilon\rangle \rightarrow_{x_1} \ldots \rightarrow_{x_{m-1}} \langle \epsilon, \sigma_m, \alpha\rangle$ can be derived in $\mathsf{Trans}'_{\mathcal{A}'}$. We thus have $\mathcal{O}^{\mathcal{A}}(\pi_1)(\sigma) = \sigma_m$. From the fact that only action executions may change the belief base, and the fact that α are the actions executed over the transition sequence, we can then conclude that $\mathcal{O}^{\mathcal{A}}(\alpha)(\sigma) = \sigma_m$. A similar line of reasoning can be followed if $\mathcal{T}'(\alpha)(\sigma)$ is not defined. □

5 Conclusion and Future Research

In this paper, we presented two formal approaches for generating and executing the plans of cognitive agents and discussed their characteristics. We explained how these approaches can be used to define the semantics of programming languages for cognitive agents in terms of operational semantics. The relation between these approaches is investigated and formally established as a theorem. The presented theorem shows that the behavior of plan generation agents is "included" in the behavior of plan execution agents.

However, for reasons simplicity, many simplifying assumptions have been introduced which make the presented approaches too limited to be applied to real cognitive agent programming languages. Future research will thus concern extending the results to more elaborate versions of the presented agent programming frameworks. Also, the characteristics of special cases will have to be investigated such as the case where there is only one extension of a plan generation agent. Finally, the notion of coherence between plans is not explored and left for future research.

References

1. G. Antoniou. *Nonmonotonic Reasoning.* Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1997.

2. M. Dastani and L. van der Torre. Programming BOID-Plan agents: deliberating about conflicts among defeasible mental attitudes and plans. In *Proceedings of the Third Conference on Autonomous Agents and Multi-agent Systems (AAMAS'04)*, pages 706–713, New York, USA, 2004.
3. M. Dastani, M. B. van Riemsdijk, F. Dignum, and J.-J. Ch. Meyer. A programming language for cognitive agents: goal directed 3APL. In *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNAI*, pages 111–130. Springer, Berlin, 2004.
4. J. de Bakker. *Mathematical Theory of Program Correctness*. Series in Computer Science. Prentice-Hall International, London, 1980.
5. M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A formal specification of dMARS. In *ATAL '97: Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, pages 155–176, London, UK, 1998. Springer-Verlag.
6. M. Georgeff and A. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, 1987.
7. G. d. Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
8. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
9. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, 1992.
10. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
11. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: a BDI reasoning engine. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin, 2005.
12. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. van der Velde and J. Perram, editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.
13. R.E.Fikes and N.J.Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
14. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
15. J. Thangarajah, L. Padgham, and M. Winikoff. Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 2003.
16. J. Thangarajah, M. Winikoff, L. Padgham, and K. Fischer. Avoiding resource conflicts in intelligent agents. In F. van Harmelen, editor, *Proceedings of the 15th European Conference on Artifical Intelligence 2002 (ECAI 2002)*, Lyon, France, 2002.
17. M. B. van Riemsdijk, F. S. de Boer, and J.-J. Ch. Meyer. Dynamic logic for plan revision in intelligent agents. In J. A. Leite and P. Torroni, editors, *Computational logic in multi-agent systems: fifth international workshop (CLIMA'04)*, volume 3487 of *LNAI*, pages 16–32, 2005.

A Functional Program for Agents, Actions, and Deontic Specifications*

Adam Zachary Wyner

King's College London
London, UK
adam@wyner.info

Abstract. We outline elements of the *Abstract Contract Calculator*, a prototype language implemented in Haskell (a declarative programming language) in which we simulate agents executing abstract actions relative to deontic specifications. The deontic specifications are *prohibition*, *permission*, and *obligation*. The concepts of deontic specifications are derived from Standard Deontic Logic and Dynamic Deontic Logic. The concepts of abstract actions are derived from Dynamic Logic. The logics are declarative, while the implementation is operational. In contrast to other implementations, we have articulated and productive violation and fulfillment markers. Our actions are given with explicit action preconditions and postconditions, and we have deontic specification of complex actions. We implement inference in the *Contrary-to-Duty Obligations* case, which has been a central problem in Deontic Logic. We also distinguish *Contrary-to-Duty Obligations* from obligations on sequences, which has not previously been accounted for in the literature. The language can be used to express a range of alternative notions of actions and deontic specification. We use it to to model and simulate multi-agent systems in which the behavior of an agent is guided by deontic specifications on actions.

1 Introduction

We present an overview of the *Abstract Contract Calculator* (ACC) written in Haskell, which is a functional programming language (cf. Wyner (2006) for the code and documentation for the ACC). The ACC processes the deontic notions of *prohibition*, *permission*, and *obligation* applied to complex, abstract actions. As an intuitive example, suppose *Bill is obligated to leave the room.* We have a *deontic specification* "obligated" applying to an *agentive action* "Bill's leaving the room". We call sets of such expressions *Contract States.* Informally, were Bill to leave the (given) room, he would have violated the obligation to leave the

* Copyright ©2006 Adam Zachary Wyner. This paper was prepared while the author was a postgraduate student at King's College London under the supervision of Tom Maibaum and Andrew Jones, which was funded by a studentship from Hewlett-Packard Research Labs, Bristol, UK. The author thanks Tom, Andrew, and HP Labs for their support and advice. Errors rest with the author.

room. Consequences may follow from the fact that he has violated his obligation. For example, he may then be obligated to pay a fine. The objective of the implementation is to abstractly model deontic specification of agentive actions as well as to simulate the behavior of agents executing actions relative to a contract state.

It is outside the scope of this paper to provide the complete implementation or formalization. Rather, we introduce basic, crucial elements and invite the reader to investigate the language further. In addition, actions and deontic notions have been extensively discussed in the Deontic Logic and Dynamic Logic literature (cf. Lomuscio and Nute (2004) and Wieringa and Meyer (1993), Harel (2000), Meyer (1988), and Khosla and Maibaum (1987)). We indicate some of our key sources. However, in contrast to these declarative works, we focus on operationalizing the notions. Therefore, we do not outline the logics. Indeed, in Wyner (2006), we have identified a range of fundamental questions with some of these logics.

The outline of the paper is as follows. We first discuss the context of our research as well as central issues along with how we address them. We turn to the implementation, which is largely presented conceptually and with fragments of Haskell code. The implementation has two aspects. First, it is a *programming tool* in that it allows alternative notions of deontic specification on agentive actions to be systematically examined and animated. Thus, one can develop and focus on a preferred interpretion of the deontic concepts. Second, having fixed an interpretation, the tool enables one to abstractly simulate environments in which agents behave relative to actions, sets of deontic specifications on actions, and how such sets change. It is intended to be used for simulation and modelling of Multi-Agent systems where deontic specifications govern the behavior of individuals or collectives of agents (see Gilbert and Troitzsch (2005) for a discussion of social science simulations). In the final two sections, we touch on other proposals for implementing deontic notions, and then we mention several aspects of the implementation which were not discussed in this paper as well as mention several aspects left for future research.

2 Background Context

The intial objective of our study was to define a formal language which is suitable for the formation, execution, and monitoring of legal contracts in a multi-agent system. Thus, our approach keeps in mind applied and empirical issues. Among the key issues, we wanted to simulate the behavior of agents with respect to deontic specifications on actions. In addition, we wanted to model how deontic specifications can *change* over time. For instance, if we have a deontic specification such as *Bill is obligated to deliver five pizzas*, we want to be able to determine the conditions under which Bill *violates* or *fulfills* this obligation. In addition, we want to determine what follows in either case. Furthermore, we want to define under what conditions can we eliminate Bill's obligation. In general, how could we operationalize deontic specifications on actions such that they could be used to *guide* agentive behavior? Thus, the research is an application of deontic reasoning.

For a theoretical underpinning, we focussed on the analyses of deontic notions, particularly those with a dynamic component (cf. Meyer (1988), Khosla and Maibaum (1987), but cf. Carmo and Jones (2001) for a non-dynamic theory). Strictly put, the implementation does not implement a particular deontic logic. We found available logics to be unsuitable for a variety or reasons (cf. Wyner (2006)). Instead, we provide a language in which different logics could be operationalized, though we make some specific suggestions.

We have implemented our system in Haskell, which is a functional programming language. Speaking broadly, functional programming languages implement the *Lambda Calculus.* It is a programming language which is particularly well suited to computational semantics (cf. Doets and van Eijck (2004) and van Eijck (2004). For a comparison to Prolog, see Blackburn and Bos (2005)).

3 Driving Issues

The implementation is driven by four interlocking issues: compositional and productive flags which signal violation or fulfillment of a deontic specification; negation of an action as *antonym* or *opposite*; complex actions, particularly sequences; and the Contrary-to-Duty paradox. In the following, we briefly outline the problems and our solutions, which we find again in the implementation.

3.1 Violability

In our view, the key concept of the deontic notions is that of *violability.* Logical or operational representations ought then to have violation (or fulfillment) markers in the formal language such that one can reason further with them (either for recovery or other processes). Thus, bad behavior is *marked* and *reasoned with* rather than ruled out (cf. Anderson and Moore (1957), Meyer (1988), and Khosla and Maibaum (1987)). We do not adopt the approach of recent proposals which use the deontic notions to *filter out* or to *prioritize* actions (cf. Garcia-Camino et. al. (2005) and Aldewereld et. al. (2005)).

In recent proposals of Standard Deontic Logic (Carmo and Jones (2001)) or Dynamic Deontic Logic (Khosla and Maibaum (1987) and Meyer (1988)), we find a distinguished *proposition* which is used to mark that a deontic specification has been violated. We use the marker for further reasoning or reactive behavior. The richer the structure of the marker, the subtler the ways it can be used (for logical proposals along these lines, cf. van den Meyden (1996, Meyer and Wieringa (1993), and Kent, Maibaum, and Quirk (1993)).

In Wyner (2006), we have argued that the markers for deontic specification on complex actions have to be productively and compositionally derived from the agent, the deontic specification, the input actions, and the mode of combination of the actions. We also argued that in order to calculate the conditions under which an *obligation* is *violated*, we need a lexical semantic function to calculate action negation. We discuss this in the next section. We should also mention that in Wyner (2006), we argue that *temporal* specifications are *not* essential to deontic specifications on actions. To our knowledge, these claims and supporting arguments are novel. They serve to distinguish our analysis and implementation.

3.2 Action Opposition and Deontic Specification

One key component of our analysis is the calculation of actions in opposition. Suppose *Bill is obligated to deliver pizzas for an hour.* It is intuitively clear that *some* actions count toward fulfilling the obligation and *some other* actions count towards violating the obligation. Furthermore, *not just any activity which is not itself an action of delivering pizzas counts toward a violation.* Indeed, some actions which Bill executes are *deontically underspecified.* If this were not the case, then *anything* Bill does other than delivering pizzas leads to a violation. More formally, set-theoretic complementation is not the appropriate notion for action negation *in our domain of application*, for it would imply that at any one time, the agent can either violate the obligation or fulfill it. There would be no actions which are deontically underspecified. This is unreasonable for agents executing contracts over time. Instead, we need some means to calculate the *opposite actions* with respect to the particular input action, leaving other actions underspecified.

In general, we want to be able to calculate the *relevant* opposite of an action, if there is one. While action opposition in natural language is rather unclear, we use *abstract actions* with respect to which we can define action opposition. Suppose α, β, and γ are abstract actions; we make these clearer in the implementation. For an action, say α, from the domain of actions, we can calculate the opposite action (given well-formedness conditions), say it is β. We can say that γ is not in any relation of opposition to either α or β. Thus, if a complex action is obligatory, we can determine what specific actions fulfill the obligation, what actions violate it, and what actions are deontically underspecified.

Other problems arise where we deontically specify complex actions. For instance, suppose we have the complex action combinator for *sequence*, where one action follows another. (α;β) represents α followed by an execution of β. Let us make this sequence obligatory: **Obligated**(α;β). Of this obligation, we want to know: What is the mark of violation or fulfillment of this obligation? Under what conditions do the marks appear? Intuitively, the mark of violation ought to indicate that the *sequence* per se has been violated (similarly for fulfillment). Thus, we need some means to define for any well-formed sequence of actions the violation marker for that sequence. Similar points can be made with respect to the other complex action combinators. In addition, we have to consider when the violation marker arises. For example, the sequence is violated where α is first executed, and then β is *not* executed, but not necessarily where β is executed before the execution of α.

In general, we have to be able to productively calculate, for any well-formed complex action, the compositional value of the violation (or fulfillment) marker. In turn, this implies that we have to calculate the *opposite* of any complex action. Thus, the lexical semantic rules must apply productively; it is not feasible to have a listing of every complex action and its opposite. Productivity and compositionality are also crucial to handle *novel actions*, which are new basic action that we introduce to a particular system. We do not want reasoning and action execution to hang when it is fed novel input.

So far as we know, the importance of productivity, compositionality, or lexical semantic opposition have not been recognized in the deontic logic literature.

3.3 Contrary-to-Duty Obligations

Contrary-to-Duty (CTD) Obligations have been a central problem in Deontic Logic (cf. Carmo and Jones (2001), which claims that it is the defining problem). Thus, an implementation ought to provide for it. CTDs are those cases where a secondary obligation arises in a context where a primary obligation has been violated. In other words, having violated one obligation, one incurs another obligation. For example, if one is obligated to return a book by a specific time, then (given the rules of a particular library), one may be obligated to pay a fine. Such cases are key to legal reasoning and a case of context-dependent reasoning (cf. Carmo and Jones (2001)). We have argued (Wyner (2006)) that violation and fulfillment markers are key to distinguish a CTD case from the case where the primary obligation *changes*. In other words, it is key that the action introduces a violation marker. In virtue of this marker, the secondary obligation is introduced.

3.4 Obligations on Sequences versus Sequences of Obligations

In Wyner (2006), we have argued for a distinction between obligations on sequences and sequences of obligations, contra Meyer (1988) who conflates them (Khosla and Maibaum (1987 mention the distinction, but do not elaborate). For example, a sequence of obligations is: one is obligated to do α and then one is obligated to do β. In contrast, one could be obligated to do α followed by β. The difference is in terms of the *violation* conditions. For a sequence of obligations, each obligatory action introduces its own violation marker. For an obligation on a sequence, failure to execute part of the sequence introduces a violation marker on the sequence *per se*. This highlights the crucial role of productive, compositional markers.

To create these richer markers, we provide a richer structure for complex actions. For example, given a sequence of $\alpha;\beta$, the structure distinguishes the input actions α and β, the resultant action (suppose) γ, and the mode of formation, which is the sequence operator. Given the definitions of basic actions, the complex action operators are given functional definitions. With this, we may define a deontic specification to apply to different *parts* of the complex action relative to the complex action operator. This allows us to define *families* of deontic specifications. For example, we can define three versions of obligations on sequences: in one, the obligation distributes to each component action; in another, the obligation applies to the *collective* action; in another, the obligation applies so as to allow interruptable obligation specifications. We can map out the logical space of possibilities. This allows us a very fine-grained, more accurate analysis. From these alternatives, we can chose that which best suits the purposes of the implementation. In our domain of application, the latter notion seems most important, and it depends on complex markers which arise in a given order.

4 An Overview of the Implementation

In the following subsections, we present highlights of the modules, necessarily skipping many details. *States of Affairs* are lists of propositions along with indices for worlds and times. *Basic Actions* are essentially functions from States of Affairs to States of Affairs. *Lexical Semantic Functions* allow us to *calculate* actions in specified lexical semantic relations such as *opposite*. These functions help us define the consequences of deontically specified actions. *Deontic Operators* apply to actions to specify what actions lead to States of Affairs in which fulfillment or violation is marked relative to the action and agent. We call such a specification a *Contract Flag State*. We implement reasoning for *Contrary-to-Duty Obligations* by modifying contract states relative to violation or fulfillment *flags*. We end with a presentation of complex actions.

4.1 States of Affairs

We construct many of our expressions from basic Haskell types for strings `String`, integers `Int`, and records, which are labels associated with values of a given type. In terms of these, we have several derived types.

Definition 1.
```
type PropList = [String]
type World    = Int
type Time     = Int
type SOA      = Rec (properties :: PropList, time :: Time,
                           world :: World)
type DBSoas   = [SOA]
```

Our atomic propositions are of type *String* such as *prop1* and *prop2*. Prefixing a string with *neg-* forms the negation of a proposition, and we have a *double negation elimination* rule. Lists of propositions, of type *PropList*, form the properties which define the properties which hold of a state of affairs. We can filter the lists for consistency. This means that we remove from the model any list of properties which has a proposition and its negation such as *[prop1, neg-prop1]*. Filtering serves to *constrain* the logical space of models under consideration and used for processing. For our purposes, we do not have complex propositions other than negation. Nor do we address inference from propositions at the level of contexts.

States-of-Affairs, which are of type *SOA*, are records comprised of a list of properties along with indices for world and time. An example SOA is:

Example 1. (properties = [prop1, prop7, prop5, neg-prop3],
time = 2, world = 4)

Lists of expressions of type *SOA* are of type *DBSoas*. These can be understood as *alternative states of affairs* or *possible worlds*.

4.2 Basic Actions

An *action* is of a record of type *Action*, which has fields for a *label* of type *String*, preconditions *xcond* of type *PropList*, and postconditions *ycond* of type *PropList*. An action is used to express *state transitions* from SOAs where the preconditions hold to SOAs where the postconditions hold. An action with an agent is of type *AgentiveAction*, which is a record with fields for an action and an *Agent* of type *String*. A list of agentive actions is of type *DBAgentiveAction*.

Definition 2.

```
type  Action             =  Rec (label :: String,
                                 xcond :: PropList,
                                 ycond :: PropList)
type  DBAction           =  [Action]
type  Agent              =  String
type  AgentiveAction     =  Rec (action :: Action,
                                 agent :: Agent)
type  DBAgentiveAction   =  [AgentiveAction]
```

An example of an agentive action is:

Example 2.

```
(action = (label = Action6,
           xcond = [prop1, prop7, prop5],
           ycond = [prop3, neg-prop4, neg-prop6]),
 agent = Jill)
```

This represents an *abstract agentive action*, which contrasts with agentive actions found in natural language such as *Jill leaves*. We work exclusively with abstract agentive actions since we can explicitly work with the properties which exhaustively define them. It is harder to do so with natural language expressions since it is not clear that we can either explicitly or exhaustively define them in terms of component properties. Nonetheless, we can refer to the natural language examples where useful.

The function *doAgentiveAction* in **Definition 3** takes expressions of type *SOA* and *AgentiveAction* and outputs an expression of type *SOA*.

Definition 3. *type doAgentiveAction :: SOA → AgentiveAction → SOA*

In the definition of the function (not provided), an action can be executed so long as the preconditions of the action are a subset of the properties of the SOA with respect to which the action is to be executed. Following execution of the action, the postconditions of the action hold in the subsequent context, and the time index of the resultant SOA is incrementally updated (in this paper, we do not manipulate the world index). Further constraints on the execution of the well-formed transitions are that the properties of the resultant SOA must be *consistent* (no contradictions) and *non-redundant* (no repeat propositions). In addition, we *inertially maintain* any properties of the input *SOA* which are not otherwise changed by the execution of the action.

In (3), we have an example.

Example 3.
```
input> doAgentiveAction
  (properties = [prop1, neg-prop3, prop5, prop7],
    time = 2, world = 4)
  (action = (label = Action6,
    xcond = [prop1, prop5, prop7],
    ycond = [prop3, neg-prop4, neg-prop6]),
    agent = Jill)
output> (properties = [prop1, prop3, neg-prop4,
  prop5, neg-prop6, prop7],
  time = 3, world = 4)
```

4.3 Lexical Semantic Functions

For the purposes of deontic specification on agentive actions, we define lexical semantic functions. These functions allow us to *functionally* (in the mathematical sense) determine actions in specified relationships. This is especially important for the definition of *obligation*, where we want to determine which *specific alternatives* of a given action induce violation. One observation we want to account for is the following. Informally, if it is obligatory for Jill to leave the room, then Jill would violate the obligation by remaining in the room. On the other hand, if it is obligatory for Jill to remain in the room, then Jill would violate the obligation by leaving the room. In other words, we see a *reciprocal* relationship between actions in opposition. Furthermore, notice that if Jill's leaving the room is obligatory, then the action which fulfills the obligation and the action which violates the obligation *must both be executable* in the same *SOA*. This means that the actions have *the same precondition properties*. While the natural language case provides the intuitions behind the functions, we implement them with respect to our abstract actions. We only provide a sample of the lexical semantic functions (see Wyner 2006 for further discussion).

Let us suppose a (partial) lexical semantic function *findOpposites*, which is essentially a function from *Action* to *Action*. For processing, it takes a lexicon and some constraints. For example, suppose *findOpposites* applied to the action labelled *Action6* yields *Action7* and vice versa. While there are many potential implementations of action opposition, we have defined the function *findOpposites* such that it outputs an action which is the same as the input action but for the negation of one of the postcondition propositions. This closely models the natural language example of the opposition between *leave* and *remain*. As an illustration, we have the following:

```
Example 4.   input> findOpposites   (label = Action6,
                                    xcond = [prop1, prop7, prop5],
                                    ycond = [prop3, neg-prop4, neg-prop6])
             output> (label = Action7, xcond = [prop1, prop7, prop5],
                                    ycond = [prop3, neg-prop4, prop6])
```

Three things are important about the function *actionOpposites* for our purposes. First, we can calculate *specific alternative actions* which give rise to violations. As discussed earlier, it is unintuitive that just *any action* other than the obligated action should give rise to violation. Second, as a calculation, we can find an opposite for any action *where the lexical structure allows one.* For the purposes of deontic specification, it need *not* be the case that every action *has* an antonym (although one could define a function and lexical space to allow this). Crucially, this holds for atomic as well as complex actions. And finally, the function *actionOpposites* is defined so as to provide reciprocal actions; that is, the opposite of *Action6* is *Action7* and vice versa. Thus, the function closely models the natural language case discussed above.

4.4 Deontic Specifications

The previous three subsections are components of deontic specifications on actions, which we model on the following intuition. Suppose an agent *Jill* is obligated to delivery a pizza. This implies that were she to deliver the pizza, in the context after the delivery of the pizza, we would want to indicate that *Jill* has delivered the pizza. Moreover, by doing so, she has fulfilled her obligation *with respect to her obligation to deliver the pizza.* On the other hand, suppose *Jill* were not to deliver the pizza, which is the opposite of delivering the pizza. In this case, we should indicate in the subsequent context that Jill that has not delivered the pizza. Furthermore, by doing so, she has *violated her obligation with respect to delivering the pizza.* We assume there are *deontically underspecified* actions as well. For example, if *Jill* eats an apple, which she could do concurrently over the course of delivering the pizza or not delivering the pizza, it may be that she does not incur a violation or fulfillment flag relative to that action. While it is possible that we use a *fixed* list for some cases to determine when violation markers arise, this will not work for complex actions or novel actions, which are those actions that are not already prelisted in a lexicon.

To define the deontic specifications, we provide a type *ContractFlag.* This type is a record having fields for: the action which is executed (indicated by the label), the deontic specification on the action (i.e. *obligated*, *permitted*, or *prohibited*), the action which is deontically specified (indicated by the label and which can be distinct from the action that is executed), whether execution of the action flags for violation or fulfillment, and the agent which executes the action. Lists of contract flags are of type *ContractFlagState.*

Definition 4. *type ContractFlag = Rec (actionDone::String,*
deonticSpec::String, onSpec:: String,

```
valueFlag::String, agent::Agent)
type ContractFlagState = [ContractFlag]
```

The violation and fulfillment flags, which are *String* types that are values of *valueFlag*, are key in reasoning what follows from a particular flag. In other words, that an agent has violated an obligation on an action may imply that the agent *incurs* an additional obligation. Indeed, such reasoning is central to legal reasoning. This is further developed in the section below on *Contrary-to-Duty Obligations*.

A deontic specifier such as *obligated* is essentially a function from an *AgentiveAction* to a *ContractFlagState*. A list of actions *DBAction* and propositions *PropList* are also input for the purposes of code development.

Definition 5. *type obligatedCompFlag :: AgentiveAction →*
DBAction → [PropList] → ContractFlagState

In **Definition 6**, we give a sample of Haskell code which calculates a *ContractStateFlag* relative to an input agentive action *inAgentiveAction* (along with a lexicon and compatibility constraints). Expressions of the form *#label list* return the *value* associated with given the *label* found in the *list*. Expressions of the form [x | x ← P] are *list comprehensions* in Haskell; they are analogous to the set-builder notation of set theory, where for S = {x + 2 | x ∈ {1,. . . ,5} ∧ odd(x)}, the result is S = {3, 5, 7}. List comprehension works much the same way, but using lists rather than sets.

We discuss the code relative to the line numbers in **Definition 6**. Lines 1-2 constitute a *guard* on the function: if the action from the input agentive action *has* an opposite (i.e. is a non-empty list), only then do we return a non-empty *ContractStateFlag* list. Otherwise, we return the empty list (line 14). This reflects the conceptual point that there can only be obligations on an action where the obligation can be violated (cf. Wyner 2006). Thus, where we return a non-empty list, there is some action in opposition to the input action. In lines 3-7, we create a list of type *ContractState* which represents the *fulfillment* of the obligation on the action. In lines 7-13, we find the opposite to the input action and use it to create a list of type *ContractState* which represents the *violation* of the action. We use *++* to conjoin these to lists to produce a list of type *ContractFlagState*.

Definition 6. *obligatedCompFlag inAgentiveAction inDBAction inComp*

```
1     | ((findOpposites (#action inAgentiveAction)
2        inDBAction inComp) /= []) =
3          ([ (actionDone=(#label (#action inAgentiveAction)),
4             deonticSpec="Obligated",
5             onSpec=(#label (#action inAgentiveAction)),
6             valueFlag="Fulfilled",
7             agent=(#agent inAgentiveAction))] ++
8          [(actionDone=(#label x), deonticSpec="Obligated",
9             onSpec=(#label (#action inAgentiveAction)),
10            valueFlag="Violated",
                       agent=(#agent inAgentiveAction))
                         | x ← (findOpposites
                         (#action inAgentiveAction) inDBAction [])])
                | otherwise = []
```

To illustrate, let us assume that when we apply *obligatedCompFlag* to an agentive action labelled *Action6* with agent *Jill*. The output is:

Example 5. [(actionDone = Action6, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Violated)]

This is of type *ContractStateFlag*. It indicates that were *Jill* to execute *Action6*, then *Jill* would have fulfilled her obligation on *Action6*. On the other hand, were *Jill* to execute *Action7*, then *Jill* would have violated her obligation on *Action6*.

As lists of records, we can manipulate them. For example, we can add to or subtract from contract states. For example, the following represents Jill's obligation with respect to *Action6* and Bill's prohibition with respect to *Action9*.

Example 6. [(actionDone = Action6, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Fulfilled),
(actionDone = Action7, agent = Jill, deonticSpec = Obligated,
onSpec = Action6, valueFlag = Violated),
(actionDone = Action9, agent = Bill, deonticSpec = Prohibited,
onSpec = Action9, valueFlag = Violated)]

Manipulations of *ContractStateFlag* expressions are crucial for *modelling* contract change, which is key to the analysis and implementation of Contrary-to-Duty Obligations.

4.5 Contrary-to-Duty Obligations

To model reasoning for CTDs, we enrich our States-Of-Affairs to include expressions of type *contractFlagState* as well as *histories* of type *history*. Histories are lists of records of what was done, when, by whom, and whether it counts as a fulfillment or violation relative to a deontic specification. Such records are of type *HistoryFlag*. They are much like *ContractState* expressions, but record the world and time at which the action is executed. An important difference between *HistoryFlag* and *ContractStateFlag* expressions is in *how they are processed*. This is further developed below.

Definition 7. *type HistoryFlag = Rec (actionDone::String,
deonticSpec::String, onSpec:: String,
valueFlag::String, agent::Agent,
world::World, time::Time)
type History = [HistoryFlag]*

Our SOAs are enriched with both a *ContractFlagState* and a *History*.

Definition 8. *type SOAHistorical = Rec (properties::PropList,*
actionDone::String, history::History,
contractFlagState::ContractFlagState,
world::World, time::Time)

Actions are executed relative to a *SOAHistorical*. Action execution *doAgentiveActionSOAHist* is essentially a function from *SOAHistorical* to *SOAHistorical*. We illustrate this informally below.

Let us suppose the following is the input *SOAHistorical* to *doAgentiveActionSOAHist*. Notice that the history is empty, which means that there is no evidence that an action has been executed.

Example 7.
```
(contractFlagState =
    [(actionDone = Action6, agent = Jill,
      deonticSpec = Obligated, onSpec = Action6,
      valueFlag = Fulfilled),
    (actionDone = Action7, agent = Jill,
      deonticSpec = Obligated, onSpec = Action6,
      valueFlag = Violated),
    (actionDone = Action9, agent = Bill,
      deonticSpec = Prohibited, onSpec = Action9,
      valueFlag = Violated)],
history = [],
properties = [prop1, prop7, prop5, neg-prop4, neg-prop6],
time = 2, world = 7)
```

Suppose that Jill does execute *Action7* with respect to this *SOAHistorical*. This means that we should indicate that Jill has violated her obligation. Thus, in the *history* of the subsequent *SOAHistorical*, we record that Jill executed *Action7*. We also record that this action violates Jill's obligation to execute *Action6*, as well as the world and time stamp where the violation occurred. We also see that the time of the *SOAHistorical* is updated. The properties are updated as well.

Example 8.
```
(contractFlagState =
    [(actionDone = Action6, agent = Jill,
      deonticSpec = Obligated, onSpec = Action6,
      valueFlag = Fulfilled),
    (actionDone = Action7, agent = Jill,
      deonticSpec = Obligated, onSpec = Action6,
      valueFlag = Violated),
    (actionDone = Action9, agent = Bill,
      deonticSpec = Prohibited, onSpec = Action9,
      valueFlag = Violated)],
history = [(actionDone = Action7, agent = Jill,
    deonticSpec = obligated, onSpec = Action6,
    time = 2, valueFlag = Violated, world = 7)],
properties = [prop1, prop7, prop5, prop3, neg-prop4, prop6],
time = 3, world = 7)
```

The next step in the implementation of CTDs is to allow contract state modification *relative to actions which have been executed in the history.* Recall from the discussion of CTDs that we only want a secondary obligation to arise *in a context where some other obligation has been violated.* In other words, if a particular violation of an obligation is marked in the *History*, we want a secondary obligation to be introduced into (or subtracted from) the *ContractStateFlag* of the *SOAHistorical.* For example, suppose Jill is obligated to leave the room. If Jill violates this obligation (by remaining in the room), then she incurs a secondary obligation to pay £5 to Bill. On the other hand, if Jill fulfills her obligation, then she incurs a secondary permission to eat an ice cream. The secondary obligations or permissions only arise in cases where a primary obligation has been violated or fulfilled.

To implement this, we have to examine whether a particular violation marker appears in the history. Second, we have to make that violation marker *trigger ContractStateFlag* modification. For instance, suppose that it is marked in the *History* that Jill has violated her obligation to do *Action6* by doing *Action7.* As a consequence of that, we modify the current contract state by removing her previous obligation and introducing an obligation on *Action11.* In such an operation, only the *ContractStateFlag* is modified. This gives the appearance of inference in a state, for there is no state change marked by temporal updating.

We have a function *doRDS*, which implements action execution for *relativized deontic specifications*; it is a function from *AgentiveActions* and *SOAHistorical* to *SOAHistorical.* It incorporates modification of the *ContractStateFlag.* Where we assume the steps just outlined to the *ContractStateFlag* in (7), a result is along the following lines:

Example 9.
```
    (contractFlagState =
        [(actionDone = Action11, agent = Jill,
          deonticSpec = Obligated, onSpec = Action11,
          valueFlag = Fulfilled),
        (actionDone = Action15, agent = Jill,
          deonticSpec = Obligated, onSpec = Action11,
          valueFlag = Violated),
        (actionDone = Action9, agent = Bill,
          deonticSpec = Prohibited, onSpec = Action9,
          valueFlag = Violated)],
    history = [(actionDone = Action7, agent = Jill,
        deonticSpec = obligated, onSpec = Action6,
        time = 2, valueFlag = Violated, world = 7)],
    properties = [prop1, prop7, prop5, prop3, neg-prop4, prop6],
    time = 3, world = 7)
```

The implementation captures the essence of the CTD problem. It models how the execution of an action relative to a *ContractFlagState* induces a modification of the *ContractFlagState*.

4.6 Deontic Specification on Complex Actions

We implement complex actions as records. Complex Actions have fields for the input actions, the complex action operator, and the result of the application of the operator to the input actions. We discuss here only the sequence operator, as it raises the more complex and interesting problems for deontic specification. We represent sequences schematically as follows.

Example 10. (inActionA = ActionA, inActionB = ActionB,
operator = SEQ, outAction = ActionC)

The *outAction* is, in this case, *function composition* of the input actions (*pace* several restrictions on well-formedness): the preconditions of *ActionC* are the preconditions of *ActionA*; the postconditions of *ActionC* are those of *ActionB* together with those of *ActionA* which remain by *inertia*; the preconditions of *ActionB* must be a subset of the postcondition properties of *ActionA*; and the postcondition properties of *ActionC* must otherwise be consistent. Our decomposition of actions into explicit preconditions and postconditions as well as our explicit construction of complex actions relative to those conditions distinguishes our approach from Dynamic Logic approaches, where there are basic actions.

In Meyer (1988), obligations on sequences are reduced to sequences of obligations on the component actions. In Khosla and Maibaum (1987), obligations on sequences are irreducible to sequences of obligations, but rather are obligations on the sequence *per se*. In Wyner (2006), we have further discussion of the significance of the difference, particularly the CTD problem. Here, we simply point out that the implementation provides ways to articulate these differences. For example, suppose *Jill* is the agent of the sequence and *ActionD* is the opposite of *ActionA* and *ActionE* is the opposite of *ActionB*. To provide the distributive interpretation of obligation in Meyer (1988), Obl_{dist}, we need two components. First, we have an initial contract state for the obligation on the first action:

Example 11. [(actionDone = ActionA, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Fulfilled),
(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Violated)]

In addition, we have a ContractStateModTrigger record which specifies that in the context where the first action has been executed (checked in the history), then the obligation on the second action of the sequence is introduced. This results in the following contract state, which specifies the fulfillment and violation cases for each of the *component* actions:

Example 12. [(actionDone = ActionA, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Fulfilled),
(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionA, valueFlag = Violated),
(actionDone = ActionB, agent = Jill, deonticSpec = Obligated,
onSpec = ActionB, valueFlag = Fulfilled),
(actionDone = ActionE, agent = Jill, deonticSpec = Obligated,
onSpec = ActionB, valueFlag = Violated)]

We might say that the obligated sequence has been fulfilled where the obligations on each action have been fulfilled and in the right order.

In contrast, we could represent Khosla and Maibaum's interpretation (1987) by applying the operator to *ActionC* with a collective interpretation of obligation, Obl_{coll}. We suppose that *ActionF* is the opposite of *ActionC*:

Example 13. [(actionDone = ActionC, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Fulfilled),
(actionDone = ActionF, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

The most interesting case is the interruptable notion of obligation on a sequence. In this case, there is a violation and fulfillment flag with respect to the *whole sequence*, and the actions must apply in a given order. We assume the following initial contract state, where we emphasize that the marker for violation is relative to the complex action *per se* and there is no marker for fulfillement of the sequence:

Example 14. [(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

The second component is the ContractStateModTrigger, which specifies that *after* execution of the first action *ActionA*, an obligation to execute the second action arises such that fulfillment of this obligation marks fulfillment of the obligation of the sequence, while violation of this obligation marks violation of the obligation on the sequence. The resulting contract state looks like:

Example 15. [(actionDone = ActionD, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),
(actionDone = ActionB, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Fulfilled),
(actionDone = ActionE, agent = Jill, deonticSpec = Obligated,
onSpec = ActionC, valueFlag = Violated),

It is in such cases that a productive and compositional analysis comes to the fore.

These examples show that there are alternative definitions which can be used to define deontic specification on complex actions. The particular definitions may be designed to suit particular purposes and interpretations. The language is thus very expressive and can be used to implement different notions of values applied to actions for the purposes of simulation in a multi-agent system. Further discussion appears in Wyner (2006).

5 Some Comparisons

There have been several recent efforts to operationalize deontic specifications. Some we have already discussed. For example, Garcia-Camino et. al. (2005) and Aldewereld et. al. (2005) appear to use deontic specifications to filter out or sort actions. We do not believe that this represents the essence of the deontic notions. Sergot (2006) uses the event calculus and only considers permissions. While we may eventually want to integrate deontic specifications into an event calculus, we would want to be clear about deontic specifications themselves; it does not seem necessary to add the additional and potentially obscuring components of the event calculus. In addition, Sergot (2006) has neither complex actions nor an analysis of the CTD problem. Boella and van der Torre (2006 present an architecture for normative systems which is similar in that deontic specifications *add* information to basic information. However, it is unclear how they implement their design, integrate complex actions, or account for the CTD problem.

6 Other Elements of the Implementation and Future Research

One key aspect of the implementation which we have not discussed here are *consistency* constraints and *implicational* relations between deontic specifications. For this, we define a notion of the *negation* of a deontic specification. We also introduce lexical relations between positive and negative deontic specifications. Further discussion appears in Wyner (2006).

We plan to enrich the structure of agents to give them some capacity to *reason* with respect to their goals, preferences, and relationships to other agents. As we want to model organizational behavior, we want to add *roles*, *powers*, a *counts as* relation between actions, and *organizational struture* to the implementation. The jural relations of *rights* and *duties* can also be incorporated into the language. While the implementation provides a significant and novel advance in the field, much yet remains to be done.

References

Aldewereld, et. al.: Designing Normative Behaviour by the Use of Landmarks. In G. Lindeman, et. al. (eds.) *Proceedings of AAMAS-05 International Workshop on Agents, Norms and Institution for Regulated Multi Agent Systems*. Utrecht, (2005), 5-18.

Anderson, A., Moore, O.: The Formal Analysis of Normative Concepts. The American Sociological Review. **22** (1957) 9-17

Boella, G., and v. d. Torre, L.: An Architecture of a Normative System. Proceedings of AAMAS'06, May 8-12, 2006, Hakodate, Hoddaido, Japan (2006)

Garcia-Camino, et. al.: A Distributed Architecture for Norm-Aware Agent Societies. In M. Baldoni et. al. (eds.) *Declarative Agent Languages and Technologies III, Third InternationalWorkshop, DALT 2005 Utrecht, The Netherlands, July 25, 2005*. London: Springer (2006)

Blackburn, P., Bos, J.: Representation and Inference for Natural Language: A First Course in Computational Semantics, Palo Alto, CA: CSLI Publications, (2005)

Carmo, J., Jones, A.: Deontic Logic and Contrary-to-duties. In D. Gabbay and Franz Guenthner (eds.) Handbook of Philosophical Logic, Dordrecht: Kluwer Academic Publishers, (2001)

Carmo, J., Jones, A.: Deontic Database Constraints, Violation, and Recovery. Studia Logica. **57** (1996) 139-165

d'Altan, P., Meyer, J.-J.Ch., Wieringa, M.: An integrated framework for ought–to–be and ought–to–do constraints. Artificial Intelligence and Law. **4** (1996) 77–111

Doets, K., van Eijck, J.: The Haskell Road to Logic, Maths and Programming, London: King's College Publications, (2004)

Dowty, D.: Word Meaning and Montague Grammar. Dordrecht, Holldand: Reidel Publishing Company (1979)

van Eijck, J.: Computational Semantics and Type Theory. Website download – http://homepages.cwi.nl/ jve/cs/, (2004)

Gilbert, N., Troitzsch, K.: Simulation for the Social Scientist, London, UK: Open University Press, (2005)

Harel, D., Kozen, D., and Tiuryn, J.: Dynamic Logic. Cambridge, MA: The MIT Press (2000)

Jones, A., Sergot, M.: On the Characterisation of Law and Computer Systems: the Normative Systems Perspective. In J.-J.Ch Meyer and R.J. Wieringa (eds.) Deontic Logic in Computer Science – Normative System Specification. Wiley (1993), 275-307

Kent, S., Maibaum, T., and Quirk, W.: Formally Specifying Temporal Contraints and Error Recovery. In Proceedings of the IEEE International Symposium on Requirements Engineering, IEEE C.S. Press, 208-215

Khosla, S., Maibaum, T.: The Prescription and Description of State-Based Systems. In B. Banieqbal, H. Barringer, and A. Pneuli (eds.) Temporal Logic in Specification. Springer-Verlag (1987) 243-294

Lomuscio, A. and D. Nute (eds.): Deontic Logic in Computer Science: Proceedings of the 7th International Workshop on Deontic Logic in Computer Science. R. Thomason (ed.), London, Springer, (2004)

Makinson, D.: On a Fundamental Problem of Deontic Logic. In P. McNamara and H. Prakken (eds.) Norms, Logics, and Information Systems. New Studies in Deontic Logic and Computer Science. IOS Press 1999 29-53

Meyden, R. v. d.: The Dynamic Logic of Permission. Journal of Logic and Computation. **6** (1996) 465-479

Meyer, J.-J.Ch.: A Different Approach to Deontic Logic: Deontic Logic Viewed as a Variant of Dynamic Logic. Notre Dame Journal of Formal Logic. **1** (1988) 109-136

Montague, R.: Formal Philosophy: Selected Papers of Richard Montague. R. Thomason (ed.), New Haven, Yale University Press, (1974)

Meyer, J.-J.Ch., Wieringa, R.J.: Actors, Actions, and Initiative in Normative System Specification. Annals of Mathematics and Artificial Intelligence. **7** (1993) 289-346

Penner, J., Schiff, D., Nobles, R. (eds.): Introduction to Legal Theory and Jurisprudence: Commentary and Materials. London, Buttersworth Law (2002)

Royakkers, L.: Representing Legal Rules in Deontic Logic. Ph.D. Thesis, Katholieke Universiteit Brabant, Tilburg (1996)

Sergot, M.: A Brief Introduction to Logic Programming and its Applications in Law. In C. Walter (ed.) Computer Power and Legal Language. Quorum Books (1988) 25-39

Sergot, M.: The Representation of Law in Computer Computer Programs. In T.J.M. Bench-Capon (ed.) Knowledge-Based Systems and Legal Applications. Academic Press (1991) 3-67

Sergot, M. and Richards, R.: On the Representation of Action and Agency in the Theory of Normative Positions. Fundamenta Informaticae. **48** (2001) 273-293

Sergot, M.: Normative Positions. In Henry Prakken and Paul McNamara (eds.) Norms, Logics and Information Systems. New Studies in Deontic Logic and Computer Science. IOS Press (1998) 289-310

Sergot, M.: A Computational Theory of Normative Positions. ACM Transactions on Computational Logic. **2** (2001) 581-622

Sergot, M.: $(C+)^{++}$: An Action Language for Modelling Norms and Institutions. technical report at http://www.doc.ic.ac.uk/research/technicalreports/2004/ DTR04-8.pdf

Wieringa, R.J., Meyer, J.: Deontic Logic in Computer Science: Normative System Specification. John Wiley and Sons (1993)

Wyner, A.Z.: Violations and Fulfillments in the Formal Representation of Contracts. ms King's College London, Department of Computer Science, submitted for the Ph.D. in Computer Science (2006)

Wyner, A.Z.: Maintaining Obligations on Stative Expressions in a Deontic Action Logic. In A. Lomuscio and D. Nute (eds.) Deontic Logic in Computer Science. Springer (2004), 258-274

Author Index