

University of Edinburgh  
School of Informatics

Automated Ontology Evolution  
Ontology Protection and Repairs Sharing

4th Year Project Report  
Artificial Intelligence and Computer Science

Agnieszka Bomersbach

2011

**Abstract:**



## Acknowledgements



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Organisation of Dissertation . . . . .	1
<b>2</b>	<b>Literature Survey</b>	<b>3</b>
2.1	Agents and Multiagent Systems . . . . .	3
2.1.1	Agent Communication . . . . .	4
2.2	Ontologies . . . . .	5
2.2.1	Ontology evolution . . . . .	6
2.2.2	Solving Ontological Mismatches . . . . .	6
2.3	Semantic Web . . . . .	7
2.4	Summary . . . . .	7
<b>3</b>	<b>Ontology Repair System</b>	<b>9</b>
3.1	Types of Repairs . . . . .	10
3.2	Surprising questions . . . . .	12
3.3	Structure of the system . . . . .	12
3.4	Flow of the System . . . . .	13
3.5	The Limitations of ORS . . . . .	15
3.6	Summary . . . . .	15
<b>4</b>	<b>Specification</b>	<b>17</b>
4.1	Main Goals . . . . .	17
4.2	Water Management Scenario . . . . .	17
4.3	Terminology . . . . .	18
4.4	Hypotheses . . . . .	19
4.5	Ontology Protection . . . . .	20
4.5.1	Types of protection . . . . .	20
4.5.2	Levels of protection . . . . .	21
4.6	Extended Agent Environment . . . . .	21
4.7	Negotiation Process . . . . .	22
4.8	Summary . . . . .	25
<b>5</b>	<b>Implementation</b>	<b>27</b>
5.1	Mark-up for Ontology Protection . . . . .	27
5.2	Extensions to the Translation System . . . . .	29
5.2.1	Translation of Protection Information . . . . .	29
5.2.2	Translation of Information Specific for SPAs . . . . .	30
5.3	Extensions to the Repair System . . . . .	30

5.4	Extensions to the Communication System . . . . .	31
5.5	Summary . . . . .	35
<b>6</b>	<b>Evaluation</b>	<b>37</b>
6.1	Experiments . . . . .	37
6.2	The Water Management Scenario . . . . .	38
6.2.1	Case A . . . . .	38
6.2.2	Case B . . . . .	41
6.3	The Shopping Ontology . . . . .	44
6.3.1	Case C . . . . .	44
6.4	Results . . . . .	46
6.5	Summary . . . . .	47
<b>7</b>	<b>Further Work</b>	<b>49</b>
7.1	Richer diagnosis . . . . .	49
7.2	Complex mismatches . . . . .	49
7.3	Dependencies between arguments . . . . .	50
7.4	Broader evaluation . . . . .	50
<b>8</b>	<b>Conclusions</b>	<b>51</b>
	<b>Bibliography</b>	<b>53</b>
<b>A</b>	<b>Ontologies</b>	<b>55</b>
A.1	The Water Management Scenario Ontologies . . . . .	55
A.1.1	ont.in . . . . .	55
A.1.2	metaOnt.in . . . . .	58
A.2	The Shopping Scenario Ontologies . . . . .	60
A.2.1	ont.in . . . . .	60
A.2.2	metaOnt.in . . . . .	63
A.3	Mark-up for ontology protection . . . . .	67
<b>B</b>	<b>Evaluation outputs</b>	<b>69</b>
B.1	Case A . . . . .	69
B.1.1	Case A1 . . . . .	69
B.1.2	Case A2 . . . . .	70
B.1.3	Case A3 . . . . .	73
B.2	Case B . . . . .	75
B.2.1	Case B1 . . . . .	75
B.2.2	Case B2 . . . . .	77
B.2.3	Case B3 . . . . .	78
B.3	Case C . . . . .	80
B.3.1	Case C1 . . . . .	80
B.3.2	Case C2 . . . . .	85





# 1. Introduction

## 1.1 Motivation

Representation of knowledge is a fluent[6] and constantly evolves to adapt to the changes of the world. This is not only true for humans: environments in which artificial agents operate are also dynamic and change rapidly. Therefore, an ontology, representing the knowledge of an agent has to be modified accordingly.

The problem arises, however, when the changes made to the ontology prevent further interaction between agents that could easily communicate in the past. The Ontology Repair System (ORS)[9] provides an innovative solution to this problem. It is an automatic tool that detects the reason of an ontological mismatch and repairs it. The problem is, however, how to decide which of the agents should repair the ontology. Which of the agents is an authority of the manner?

In the original version of the system, always the agent that formed plans was chosen to perform the repairs proposed by ORS. We believe, however, that agents should be autonomous and have the possibility to decide what parts of the ontology can be affected by the repair and what should stay unchanged.

In the open environments, like the Semantic Web, agents interact with previously unknown agents. It is often impossible to verify whether these agents are trusted or not. Modifications made to the ontology, that aim at enabling further interaction, might lead to the loss of relevant facts or addition of false information.

The aim of this report is to present an extension to ORS that provides agents with a control over the changes made to the ontology. The user can choose what parts of the ontology should be kept secure. It is also possible to define to what extent the data is protected.

We use agent negotiation to verify which of the agents should repair the ontology. Protection of the ontology determines if the agent is prepared to perform the repair.

## 1.2 Organisation of Dissertation

In **Chapter 2** we present main concepts that are relevant to this project. We introduce ontologies and discuss their role in multi-agent systems. We analyse the issue of mismatches between ontologies and present some solutions to this problem.

In **Chapter 3** we describe the functionalities of ORS and the types of repairs it can perform.

In **Chapter 4** we present the main goals of the project and the requirements that have to be met by the extended version of ORS.

In **Chapter 5** we present the design decisions we have made and the results we have obtained.

In **Chapter 6** we examine the performance of the extended system. We present the results of the experiments we have conducted.

In **Chapter 7** we discuss the limitations of our approach and possible improvements.

In **Appendices** we include the code of scenarios used for evaluation, the ORS outputs we obtained during the experiments and the glossary.

## 2. Literature Survey

In this chapter we describe these areas of Artificial Intelligence that are particularly relevant to the project. We introduce agents, multiagent systems and ontologies.

### 2.1 Agents and Multiagent Systems

An agent is a computer system that is situated in some environment and that is capable of interaction with it. Agents can perceive the environment and perform autonomous action in order to meet their objectives.

[12] presents main features of an intelligent agent:

- Reactivity - the ability to constantly interact with the environment and respond to the changes in it.
- Proactiveness - the ability to generate plans and attempt to achieve goals by taking the initiative instead of waiting for instructions.
- Social ability - the ability to communicate with other agents in order to achieve goals.
- Autonomy - the ability to act without intervention and have some control over the state and actions.

[10] describes various possibilities of agents' environments:

- Accessible vs. inaccessible - agents can or cannot obtain complete, accurate and up-to-date information about the environment's state. The majority of moderately complex environments (such as the Internet) are inaccessible.
- Deterministic vs. Non-deterministic - there is or there is not a certainty about the state that results from performing an action. Non-deterministic environments present greater problems for the agent designers.
- Dynamic vs. Static - the state does or does not change during an agent's actions. A dynamic environment has other processes operating on it, which bring changes beyond the agent's control.
- Episodic vs. Non-Episodic - the performance of an agent depends on the number of discrete episodes or only on the current episode.
- Discrete vs. Continuous - there is or there is not a fixed, finite number of actions and percepts in the environment.

Although there are situations where an agent can operate usefully by itself, the increasing interconnection and networking of computers is making such situations rare [11]. More often we deal with a system that consists of several agents that interact with one another in order to achieve their goals. This is what we call a multiagent system (MAS). Agents perform tasks on behalf of their owners or users, generally with different goals and motivations. In order to successfully interact, agents must have the ability to:

- cooperate
- coordinate
- negotiate
- compete

with one another.

### 2.1.1 Agent Communication

Agents are distributed within the MAS and differ in many aspects. Some common terminology needs to be introduced in order to be able to exchange knowledge about the environment or goals or manage interdependencies, hence communicate.

Four levels of communication should be considered:

- Message Semantics that focuses on what the message means. It is composed of:
  - Message type that gives intensionality.
  - Message content that contains information.
  - Ontology (further discussed in Section 2.2).
- Message Syntaxes that focuses on how a message is expressed. It is composed of:
  - Message Structure: Agent Communication Language
  - Content Codification: Content Language
- Interaction Protocol that focuses on how dialogues are structures.
- Transport Protocol that focuses on how messages are sent/received.

*Speech Act Theory*, firstly mentioned in [4], influenced many attempts to model communication. In this theory, communication is treated as an action that causes

some change in the world. Examples proposed by Austin are: declaring a war or “I pronounce you man and wife”.

There are three aspects of *Speech Theory*:

- *locutionary act* or *locution*, showing what is said or written (e.g. use airline’s website)
- *illocutionary act* or *illocution*, what is meant (e.g. suggest)
- *perlocutionary act* or *perlocution* which expresses the effect of the action (e.g. people visit the website and book flight tickets)

A *Speech Act* consists of the performative verb and the propositional content. For example, having the performative verb “request” and the content “close the door” yields the Speech Act: “please close the door”.

Agent communication is based on *Speech Act Theory* by using a set of pre-defined performatives. There are some pre-defined performative sets used in multiagent systems. Two most common agent communication languages are:

- KQML - the first widespread Agent Communication Language, developed by the ARPA knowledge sharing initiative. KQML is comprised of two parts: the knowledge query and manipulation language (KQML) and the content language (usually KIF). KQML consists of a performative, communication parameters, message content, content language specification and ontology specification.
- FIPA-ACL - developed by the Foundation of Intelligent Physical Agents (FIPA) to be a program of agent standards. The basic structure of FIPA-ACL is similar to the one of KQML, it consists of type of communicative act, communication actors, content, content description and conversation control. Almost any content language can be used with FIPA-ACL, but most used are KIF, RDF, DAML, OWL and FIPA-SL.

## 2.2 Ontologies

In order to enable communication, some logical theories for representing knowledge of an agent have to be introduced. These theories are called ontologies. The term ontology comes from Greek *ontos* (of being/to be) and *logos* (word). [7] specified it as an explicit specification of a conceptualization.

Ontologies not only represent knowledge, but also allow sharing the information structure between agents or people, allow knowledge reuse and make explicit the interpretations about some domain.

Ontologies have to be expressed in a machine-computable language in order to enable agents and other intelligent systems to use them for sending messages and reasoning. The language should be easy enough to simplify the task of ontology development and with computational cost reasonably low. On the other hand, the language needs to be rich enough in order to fully represent the domain.

There are several languages used for ontologies, such as DAML-OIL (DARPA agent markup language), OWL (Ontology Web Language, which is a fusion of DAML-OIL and was chosen by W3C to be a standard web language).

In this project, we shall focus on the Knowledge Interchange Format (KIF) [1]. The motivation was to have an exchange format between applications, which does not depend on their internal representation. KIF is based on First Order Logic (FOL) and has FOL's operators such as boolean values, connectives, variables and quantifiers. KIF agents can express properties and relations between things in the domain as well as general properties of a domain.

### 2.2.1 Ontology evolution

Ontologies constantly evolve in order to adapt to the changes in the environment or in goals. Differences between evolving ontologies may lead to mismatches that disable further interaction. Various approaches have been taken to address this issue.

### 2.2.2 Solving Ontological Mismatches

*Ontology merging* is a solution where one common ontology is built to be shared by interacting agents. The other idea - *ontology matching* - focuses on finding the mappings between all the terms in one ontology to some terms another one. It can be seen that both merging and matching have to be performed offline, prior to interaction. Another negative side of these solutions is the necessity to have the access to whole ontologies. In the case of a large multiagent system, like the Semantic Web, agents interact with previously unknown agents in order to perform desired tasks. Access to the ontology of another agent should be minimal, giving the possibility of hiding confidential knowledge. The other negative feature about these approaches comes from the fact that an agent's environment changes rapidly, therefore problems should be solved as soon as they are encountered.

Another approach to the problem - *ontology repair* - and the Ontology Repair System developed by [9] will be discussed in the next chapter.

## **2.3 Semantic Web**

[5]

## **2.4 Summary**



### 3. Ontology Repair System

The Ontology Repair System (ORS) brings a novel solution to the problem of ontological mismatches. It is an automated system that provides online repair of ontologies in the situation of a mismatch. It works as a plug-in to a planning agent (PA) which interacts with service providing agents (SPAs). The aim of the PA is to perform the plan and ask SPAs to perform actions leading to the goal. When the mismatch between ontologies occurs, ORS analyses the problem and gives a diagnosis. ORS then repairs the PA's ontology and replans. This interaction continues until the goal is reached.

An important aspect of ORS is that it does not have access to the ontologies of SPAs and repairs are performed only when mismatches violate the plan execution. ORS is mainly used for ontologies that come from the same source but have evolved in the meantime.

The system is written in SICStus Prolog [3] and works with KIF ontologies, such as the ones provided by Ontolingua [2].

ORS uses first-order ontologies which [9] defines as follows:

“We consider first-order ontologies that contain:

- A signature containing:
  - Definitions of predicates: their names, arity and number of arguments. Unary predicates are class definitions.
  - Details of relationships between predicates. In the case of the unary predicates, these relationships define a class hierarchy.
- A theory containing:
  - Instantiations of the predicates defined in the theory.
  - Action rules, which define the way in which performing actions as part of a plan can alter the truth values of facts in the theory. Action rules have preconditions, stating which facts must be true or false in the theory for them to be applicable, and effects, stating what facts are altered as a result of the action”.

### 3.1 Types of Repairs

Many of the mismatches that are solved by ORS concern differences in signature. The ontology of the PA might miss some details or have more than the SPA's one. There are two types of repair provided in such a situation:

- *abstraction* - performed when it is necessary to remove a detail.
- *refinement* - performed when it is necessary to add a detail.

ORS deals with mismatches between ontological objects of the same type. Only simple mismatches are treated, combinations of other mismatches are not considered. We shall now describe the repairs performed by ORS:

#### 1. Change of a predicate

- **predicate's name**

The name of the predicate changes but the arguments stay the same.

- *predicate refinement*

A predicate is changed into the more specific one.

For example, changing *Book*<sup>1</sup> into *Cookbook* gives:

$$(Book \ ?Title \ ?Price) \mapsto \\ (Cookbook \ ?Title \ ?Price)$$

- *predicate abstraction*

A predicate is generalized into its superclass.

For example, changing *Book* into *Item* gives:

$$(Book \ ?Title \ ?Price) \mapsto \\ (Item \ ?Title \ ?Price)$$

- **predicate's arity**

Number of arguments of the predicate is changed.

- *propositional refinement*

An argument is removed from the predicate.

For example, removing *?Price* from *Book* gives:

$$(Book \ ?Title \ ?Price) \mapsto \\ (Book \ ?Title)$$

- *propositional abstraction*

A new argument is added to the predicate.

For example, adding *?CreditCardPayment* into *Book* gives:

$$(Book \ ?Title \ ?Price) \mapsto \\ (Book \ ?Title \ ?Price \ ?CreditCardPayment)$$


---

<sup>1</sup>For the aid of explanation, we use a predicate:  
(*Book* ?*Title* ?*Price*)

- **argument's type**

- *domain refinement*

A predicate is changed into the more specific one.

For example, changing  $?CreditCardPayment$  into  $?VISAPayment$  in *Book* gives:

$$\begin{aligned} & (Book \ ?Title \ ?Price \ ?CreditCardPayment) \mapsto \\ & (Book \ ?Title \ ?Price \ ?VISAPayment) \end{aligned}$$

- *domain abstraction*

A predicate is generalized into its superclass.

For example, changing  $?PaymentType$  into  $?CreditCardPayment$  in *Book* gives:

$$\begin{aligned} & (Book \ ?Title \ ?Price \ ?CreditCardPayment) \mapsto \\ & (Book \ ?Title \ ?Price \ ?WayOfPayment) \end{aligned}$$

- **arguments' positions**

Arguments are switched.

For example, switching  $?Title$  with  $?Price$  gives:

$$\begin{aligned} & (Book \ ?Title \ ?Price) \mapsto \\ & (Book \ ?Price \ ?Title) \end{aligned}$$

- **predicate relationships**

Relationships between predicates form a hierarchy. More general predicates are known as the super-predicate. For example, *Item* is a super-predicate of *Book*. Suppose we have another predicate, *Present*, with the same arguments as *Book*, but without a super-predicate. For example, adding a super-predicate *Item* to *Present* changes the hierarchy.

## 2. Change of an action rule

Components of an action rule: preconditions and effects, can be added or removed. Changes made to the internal structure of them have been described above.

- **precondition**

- *precondition abstraction*

A precondition is added to the action rule.

Adding  $(Has \ ?Title \ ?Discount)$  into  $PurchaseBook$ <sup>2</sup> gives:

$$\begin{aligned} & (Book \ ?Title \ ?Price) (InStock \ ?Title) \rightarrow \\ & (Purchased \ ?Title) \end{aligned}$$

$$\mapsto$$

$$\begin{aligned} & (Book \ ?Title \ ?Price) (InStock \ ?Title) (Has \ ?Title \ ?Discount) \rightarrow \\ & (Purchased \ ?Title) \end{aligned}$$


---

<sup>2</sup>For the aid of explanation, we use an action rule  $PurchaseBook$ :  
 $(Book \ ?Title \ ?Price) (InStock \ ?Title) \rightarrow (Purchased \ ?Title)$

- **effect**

- *precondition refinement*

An effect is added to the action rule.

Adding  $(Delivered\ ?Title)$  into  $PurchaseBook$  gives:

$$(Book\ ?Title\ ?Price)\ (InStock\ ?Title) \rightarrow (Purchased\ ?Title)$$

$$\mapsto$$

$$(Book\ ?Title\ ?Price)\ (InStock\ ?Title) \rightarrow (Purchased\ ?Title)\ (Delivered\ ?Title)$$

### 3. Change of an individual

A name or a type of an individual is changed.

For example,  $StudentDiscount \mapsto PensionerDiscount$ .

## 3.2 Surprising questions

Another thing introduced in [9] is the notion of *surprising questions*. When the PA is asked for a precondition that it does not expect to be asked about, we encounter a surprising question. This situation might happen when the PA does not have the precondition in the rule of the action or when some parameters of this precondition are different, such as different arity or types of arguments. Surprising questions are a very good information about where the two ontologies differ. Often it is just enough to find the problem and repair it.

## 3.3 Structure of the system

We shall now focus on the components of the system and their functionalities:

- **Agent Environment**

This is a platform that enables agents to communicate by exchanging messages. In this way any issues concerning actions are clarified. ORS uses a simple communication protocol with two performatives: *query* and *reply*. They contain the information about: the sending agent, the receiving agent, the type of query (i.e. *request*, *question* or *truth*) and the content of the message.

- PA

This is a generic version of the Planning Agent. It has an access to its ontology and to the Prolog Planner that provides a plan leading to the goal.

- scenarios
  - \* SPAs
 

Each scenario has a set of Service Providing Agents that are responsible for different tasks.
  - \* PA's ontology
 

As the generic PA is used in the system, specific ontologies for different scenarios are provided.
- **ORS**

ORS can be seen as a plug-in to the PA that helps to repair the ontology when the mismatch occurs. It consists of four subsystems:

  - **the Translation System**

Although the central ontologies are written in KIF, there is a need to have a Prolog version of them as well as they are necessary for actions in other parts of the system. This requires the translation from KIF to Prolog.
  - **the Plan Deconstructor**

Given the plan, it produces an ontological justification for it.
  - **the Diagnostic System**

The plan provided by the planner is executed until an ontological mismatch occurs. The Diagnostic System is then called in order to detect the reason for the problem and to suggest a possible solution. It provides the Refinement System with all the necessary information to enable it to carry out the task.
  - **the Repair System**

The aim of this component is to perform the repair suggested by the Diagnostic System. Afterwrd, the replanning is necessary.

### 3.4 Flow of the System

Figure 3.1 illustrates the flow of the system. The PA is provided with the goal at the beginning of an interaction. First of all, the PA translates the ontology to the Prolog representation. It then asks the Prolog Planner for the plan leading to the goal. If it is not possible to reach the goal based on the PA's ontology, no plan is formed. In the other case, the interaction between agents begins.

For each task, that is part of the plan, the PA finds an appropriate SPA to perform it and sends a request to do so. Once the SPA receives the request, it checks whether it is possible to perform the task. Some questions might be asked

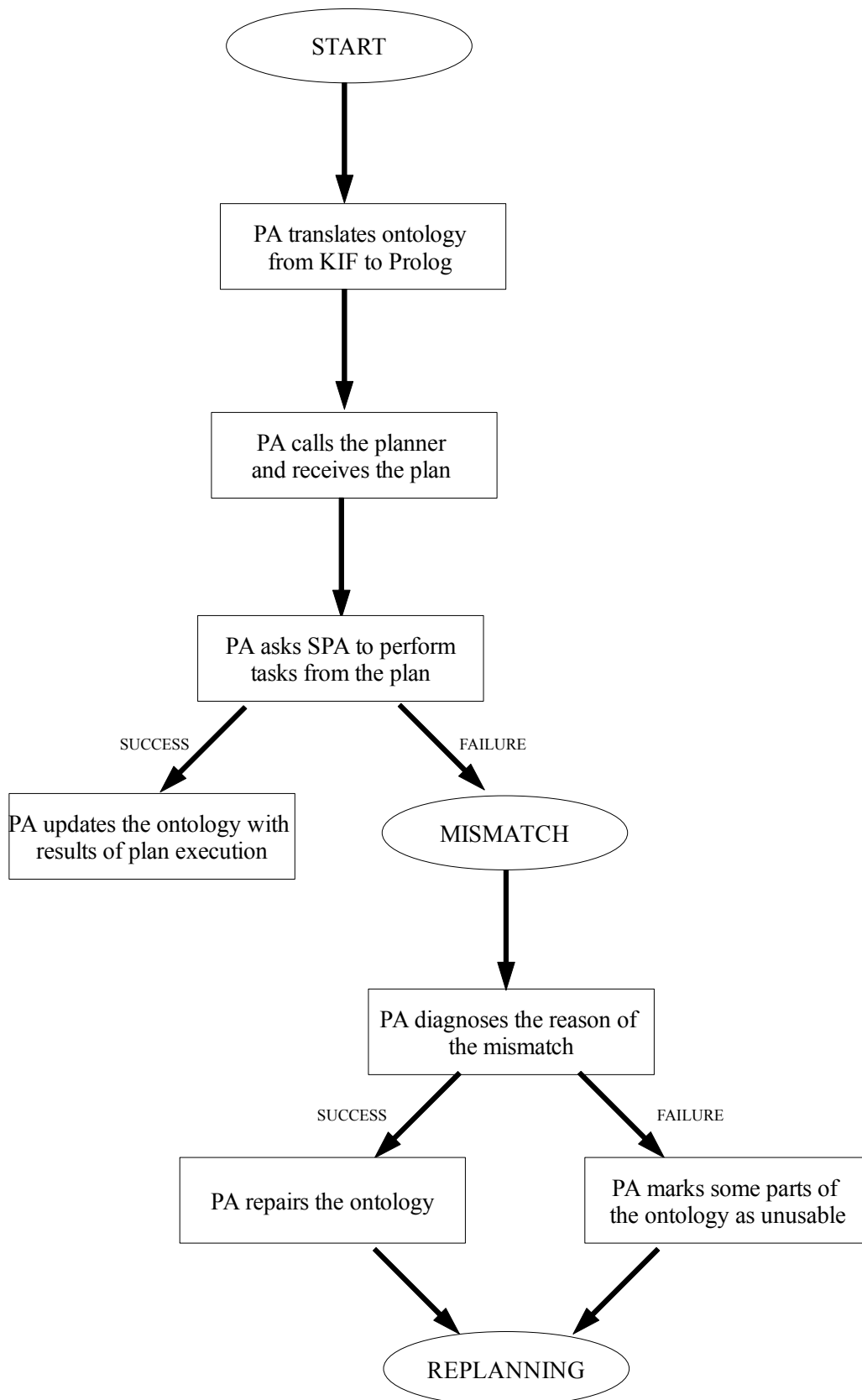


Figure 3.1: Flow of the System (adapted from [8])

to verify whether preconditions are met. If the task is performed successfully, the next part of the plan is considered. However, in the situation of a mismatch between ontologies, the Diagnostic System of ORS is called to find the reason for the problem. The Repair System is provided with the proposed repair and performs the repair whenever possible. Replanning follows any repair attempts.

Once all the actions have been performed and the goal is achieved, the interaction finishes.

### 3.5 The Limitations of ORS

In the original version of ORS, there was no negotiation about which agent should repair its ontology. It was always the PA that performed the repair proposed by the system and willingly accepted to do so. It had no control over the changes made to the ontology. In many situations this was the only way to enable further interaction.

Very often agents interact with previously unknown agents in order to fulfil the plan. These agents might have very different representations of the environment. Therefore, a lack of control over the changes of the PA's ontology might lead to the loss of relevant information. Protection of an ontology would provide information as to what ontological changes are acceptable by an agent.

ORS focuses mainly on scenarios where the PA wants the SPA to perform actions for it and therefore is prepared to do anything to enable the plan execution. In such a situation, the SPA behaves as if it was an authority of the manner. However, we cannot assume that the SPA has a more relevant version of the ontology in every situation. For example, the PA should be seen as an expert of information about itself: name, basic features, etc. Considering scenarios of cooperating agents shows that it is advantageous that the PA negotiates with other agents as to which of them should alter their ontologies.

### 3.6 Summary

This chapter has presented ORS, an automatic tool that helps an agent to repair its ontology in the situation of a mismatch between agents. It is important to understand the functionality of ORS as this project is an extension of this system. We have also discussed possible additional functionalities which we will further describe in the following chapter.



# 4. Specification

In section 3.5 we presented the limitations of ORS. We shall now discuss how we address these issues and what solutions we propose. We introduce the ontology protection as the way to provide a control over the changes in the ontology. We propose the repairs sharing as the improvement in the interaction between agents.

## 4.1 Main Goals

The main goals of this project are to:

- Give agents an option of protecting important parts of their ontologies. Provide different types of protection.
- Enable any agent to repair its ontology. Give SPAs access to the Repair System and the Translation System of ORS.
- Extend the Agent Communication System to enable exchange of messages concerning repairs.
- Use negotiation between agents to decide which of them (possibly both) should repair their ontologies.

## 4.2 Water Management Scenario

The Water Management Scenario was prepared for the evaluation of the system and will be thoroughly described in Section 6.2. We provide a brief description of it here in order to facilitate the understanding of our approach.

We analyse a system of cooperating waste water treatment plants, that store water, treat contaminants and manage water to avoid the overflow. Each plant has a set of contaminants it can treat and a capacity of water that it can store. We consider a situation with one PA:  $plant_0$  and three SPAs:  $plant_1$ ,  $plant_2$  and  $plant_3$ .

KIF ontologies for this scenario can be found in Apendix A.1.

### 4.3 Terminology

We introduce some terms that are relevant to this project:

- A **good plan** is the one that is desired, but due to a mismatch in the representation, is not performable. It two tasks from the plan have the same precondition, repair performed during the execution of the first task, prevents the execution of the next one.

For example, the PA's ontology for the water management scenario contains a predicate `treats(plantZero,x,medium)`<sup>1</sup>, stating that the plant is capable of treating a contaminant `x` with medium quality. This is a precondition of two action rules: `sendContaminatedWater`<sup>2</sup> and `treatContaminatedWater`<sup>3</sup>. If, during the execution of `sendContaminatedWater`, one of the arguments of `treats` is removed, say `Quality`, the predicate changes as follows:

$$\text{treats}(\text{plantZero},x,\text{medium}) \mapsto \text{treats}(\text{plantZero},x)$$

It is no longer possible to perform `sendContaminatedWater`, as some information is missing. The plan fails.

- By a **bad plan** we mean a plan that is performed although it should not be because it leads to an undesirable situation. The repair can cause the loss or change of information that is particularly necessary for future plans. If this happens, further planning is based on the wrong representation of the world and hence the bad plan is formed. Therefore, the outcomes of the plan can be different than expected.

For example, consider a situation following the repair described above. The PA is no longer aware of the quality of treatment it provides, hence it does not verify whether it is good enough. We can see that the quality of treatment is *medium*<sup>4</sup>. As this information is now lost, the PA can perform the treatment even when the required quality of treatment is *perfect*. Such a plan should not be executed.

---

<sup>1</sup>Definition of *treats*:

```
treats(Agent,Contaminant,Quality)
```

<sup>2</sup>Definition of *sendContaminatedWater*:

```
rule(sendContaminatedWater(Agent1,Agent2,Contaminant,Quality),
[rule4, [at(Agent1,Contaminant),treats(Agent2,Contaminant,Quality)],
[at(Agent2,Contaminant),not(at(Agent1,Contaminant))] ]).
```

<sup>3</sup>Definition of *treatContaminatedWater*:

```
rule(treatContaminatedWater(Agent1,Contaminant,Quality), [rule2,
[at(Agent1,Contaminant),treats(Agent1,Contaminant,Quality)],
[goodPerformance(Agent1)] ]).
```

<sup>4</sup>The predicate before the repair is of the form:

```
treats(plantZero,x,medium)
```

## 4.4 Hypotheses

The aim of this project is to examine the following claims:

1. **This technique avoids *bad plans* by protecting important parts of the ontology. The number of plans developed decreases but the new set of plans include only *good plans*.**

Consider an interaction between  $plant_0$  and  $plant_1$ . Each plant has a set of contaminants that it can treat, say  $c_0$  for  $plant_0$ . Suppose  $plant_1$  has a wrong representation of the set of contaminants handled by  $plant_0$ , say  $c'_0$ . We have that  $c_0 \neq c'_0$ .

Choose  $x$  such that  $x \in c'_0$  and  $x \notin c_0$ . When  $plant_1$  detects such an  $x$ , it believes that  $plant_0$  can treat it, which is not true. In the current situation,  $plant_0$  would be forced to repair its ontology and as a result of that we would get  $x \in c_0$ . All the other plants would send the water with contaminant  $x$  to  $plant_0$  and that would not send it away. The reason is that having  $x$  in  $c_0$  means that  $plant_0$  believes it is able to treat  $x$ . This would result in contaminated water being distributed. In the new situation this bad plan would not be performed in this manner, as having  $c_0$  protected would prevent  $plant_0$  from changing it.

Consider also  $x \in c_0$  and  $x \notin c'_0$ . In this case,  $plant_1$  refuses to send to  $plant_0$  contaminants it can actually handle.

Another example is concerned with the wrong representation of the capacity of the plant. Suppose the capacity of  $plant_0$  in the ontology of  $plant_1$  is  $v'_0$  and in the ontology of  $plant_0$ ,  $v_0$ . If  $v'_0 > v_0$ ,  $plant_0$  would change its capacity to  $v'_0$  when interacting with  $plant_1$  because nothing is protected. That would mean that  $plant_0$  would believe it was able to receive more water than it could actually store. This would lead to the overflow of the system. In the new situation this bad plan would not be performed, as having  $v_0$  protected would prevent  $plant_0$  from changing it.

2. **This technique provides alternative *good plans* by sharing repairs between agents.**

Consider the examples above when one of the situations,  $c_0 \neq c'_0$  or  $v_0 \neq v'_0$ , occurs. Analyse the situation when  $c_0 \neq c'_0$ . Let  $c_0$  belong to the protected part of the ontology of  $plant_0$ . It is  $plant_0$  that is mostly aware of any changes in its features, like  $c_0$ . Therefore only  $plant_0$  would have this information in the protected part and  $plant_1$  would not. When the mismatch occurs,  $plant_0$  is unable to change  $c_0$  to match to  $c'_0$ . It is possible for  $plant_1$  to change  $c'_0$ , so that  $c'_0 = c_0$ . Through negotiation, these agents

should exchange all the necessary information needed to map the ontologies and successfully perform the plan.

The situation when  $v_0 \neq v'_0$  is analogous.

## 4.5 Ontology Protection

### 4.5.1 Types of protection

Our design gives agents the possibility to keep important parts of the ontology secure. We consider several types of protection concerning the information included in the theory or the signature of the ontology:

- **Protection of a predicate**

- **whole predicate**

A protection of a whole predicate is useful in the situation when its arguments depend on one another (e.g. one represents a quantity and the other one - a unit). A good candidate from the water management scenario would be a predicate `maxVolume(plantZero,100,litres)`<sup>5</sup>, representing the maximum volume of water in the plant. A protection of a whole predicate prevents changing a single argument that depends on another one:

$$\text{maxVolume(plantZero,100,litres)} \mapsto \text{maxVolume(plantZero,100,gallons)}$$

A change in unit affects the meaning of the predicate it is hence necessary to change the quantity accordingly<sup>6</sup>. Such a protection also prevents removing the whole predicate from preconditions or effects of the action rules.

- **predicate's arity**

A protection of an arity of the predicate prevents adding further information to the predicate or removing any of its arguments, e.g. adding a quality of repair, i.e. *medium* or *perfect*, to the treats predicate, i.e.:

$$\text{treats(plantZero,x)} \mapsto \text{treats(plantZero,x,perfect)}$$


---

<sup>5</sup>A definition of *maxVolume*:

`maxVolume(Agent,Number,Unit)`

<sup>6</sup>An example is a change from `maxVolume(plantZero,100,litres)` to `maxVolume(plantZero,22,gallons)`. This requires changing two arguments of the predicate at the same time which is out of scope of the system's functionality. ORS deals with single mismatches at a time.

Consequently, an agent would believe the quality of its treatment was *perfect*, although it was not necessarily true.

- **Protection of an argument**

- **whole argument**

Prevents changing any information about the argument. The protected argument cannot be removed, switched with another argument or changed. For example, protecting quality of treatment prevents:

`treats(plantZero,x,perfect) ↦ treats(plantZero,x).`

- **argument's type**

Prevents changing the type of an argument. Suppose both of the interacting agents have the predicate `treats(plantZero,x,perfect)`, but the type of `x` is different in two ontologies:

PA: `class(x,contaminantA)`

SPA: `class(x,contaminant)`, where

`subclass(contaminantA, contaminant).`

Protection of argument's type prevents:

`class(x,contaminantA) ↦ class(x,contaminant)`

Consequently, the PA would lose properties of the more specific type.

## 4.5.2 Levels of protection

Apart from defining what to protect, it should be also possible to specify to what extent do so.

- **High protection** prevents any changes to be made to the chosen parts. There are reasons other than the loss of information that might discourage an agent from performing the repair. For example, further actions or costs might be necessary for the repair.
- **Low protection** could be used for repairs leading to lower profit or financial loss.

## 4.6 Extended Agent Environment

One of the goals of this project is to provide the extended agent environment. We aim at letting any agent perform repairs, not only the PAs, as occurred before. These changes affect different subsystems of ORS.

In the original version of ORS, all the agents had the Prolog representations but only the PA was equipped with KIF ontologies. This is a limitation of ORS that we need to address. The Repair System works with KIF ontologies, therefore it is crucial to provide the SPAs with them. Prolog representations, however, are still necessary to be used by other components of the system. In order to keep consistency, parts that are modified in one representation have to be changed accordingly in the other one. This task is carried out by the Translation System, therefore it is required to give the SPAs an access to this subsystem.

One of the major differences between the SPAs and the PA is that the latter one does not perform tasks. As the Translation System was designed to fulfil the needs of the PA, it lacks functions necessary for translation of information relevant only for SPAs. We need to express the following information in KIF and extend the Translation System to handle it by changing into the Prolog representation:

- **tasksIPerform**, which takes as an argument: a list of actions that an agent can perform
- **ask**, which takes as an argument: a list of names of preconditions that should be checked during the execution of the task
- **wait**, which takes as an argument: a list of predicates that are marked with *wait*; the list should always contain *calculation*
- **nonFacts**, which takes as an argument: a list of predicates marked as *nonFacts*

## 4.7 Negotiation Process

Figure 4.1 shows an algorithm we use for the negotiation about repairs. For every repair type, there is a set of relevant protections associated with it. For example, suppose the PA has a ternary predicate **treats(plantZero,x,perfect)** and the SPA - **treats(plantZero,x)**. If the mismatch occurs, *propositional abstraction* is proposed by the Diagnostic System as the solution to the problem. It requires the PA to remove the third argument from **treats**. The verification, whether this repair can be performed, is required. In this situation, it is necessary to check the protection of the *whole predicate* **treats** and the protection of the *whole argument* at position 3 in **treats**. We shall further describe the relationships between repairs and the types of protection in Section 5.3.

If the relevant part of the ontology is not protected, the repair is performed. Otherwise, the PA asks the SPA to change its representation instead. Together with the request, all the relevant information is sent. The repair, required from

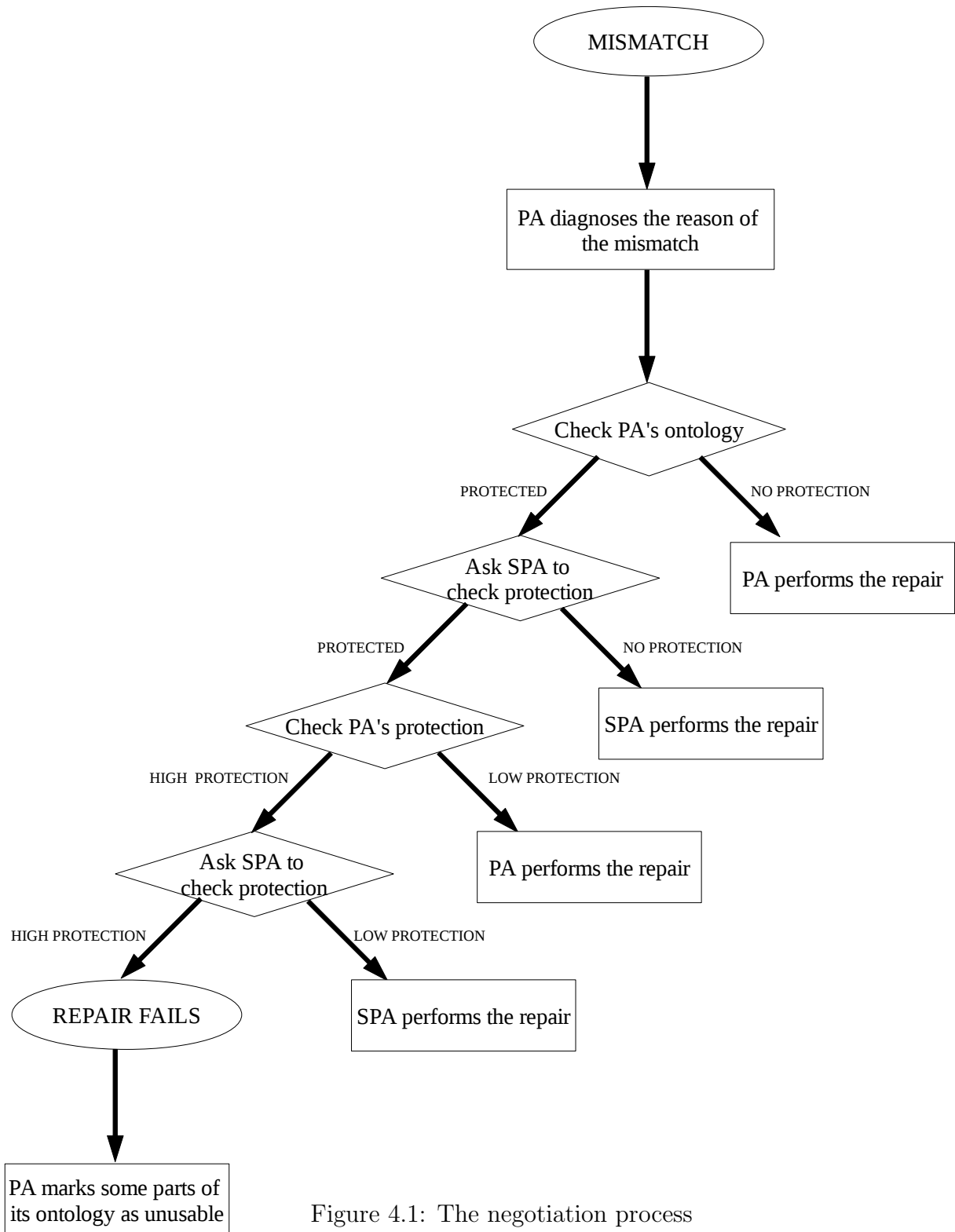


Figure 4.1: The negotiation process

the SPA, is an inverse of the repair that the PA was supposed to perform. The pairs of inverse repairs are:

- *abstraction & refinement*
- *refinement & abstraction*
- *switch arguments & switch arguments*

For example, if the PA was supposed to perform *domain refinement*, the SPA will be asked for *domain abstraction*.

Once the SPA receives the repair request, it verifies if it can change the ontology. The repair is performed whenever possible. In the case of failure, the PA checks if it can change its objections, i.e. if the protection is low. If this is the case, the repair is performed. However, if this attempt ends with failure, the SPA is asked for analogous action. The negotiation ends, if none of the agents can change their ontologies. The PA marks the SPA as unuasble for this action. The interaction with this action yields no effects, therefore it is necessary to find another agent to perform the task.

When a mismatch occurs, it is the PA that requests the Diagnostic System for a repair. If the repair cannot be performed by the PA, the SPA is requested to implement the inverse of it. All the information about the repair, available to the PA, is passed to the SPA. Therefore, there is no need for the SPA to call the Diagnostic System itself. However, this approach enables the SPA to perform *refinements* and *switching arguments* but does not support *abstractions*. More data concerning the repair is always provided in the case of diagnosis of the refinement as opposed to the abstraction. This comes from the fact that more information is required when we add a new component than when removing one. For example, we need to know a type and a value of an argument that we want to add. If the PA is supposed to perform a *propositional abstraction*, it obtains information about a predicate that is affected and a position of an argument that should be deleted. On the other hand, *propositional refinement* provides not only the name of a predicate and a position on which an argument should be added, but also the type and the value of the argument and the arity of the predicate. Suppose the PA is supposed to perform an *abstraction*, but this part of its ontology is protected, hence it asks the SPA to perform the inverse of it: *refinement*. In this case, the SPA does not have enough information to perform the repair. Therefore the refinement is not provided for the SPAs. Improving the Diagnostic System would result in obtaining richer diagnosis and possibility of addressing more complex repairs. This issue will be discussed in Chapter 7.

We observe that it is always the PA that first tries to perform the repair. The justification is that the PA forms the plan and is interested in its successful completion. One of the aims of this project is to help the PA to continue the

interaction in the situation when it refused to make any changes. In all other cases, the PA should repair its ontology.

## 4.8 Summary

In this chapter we have presented the hypotheses of the project and design decisions concerning the ontology protection and the negotiation process.



# 5. Implementation

In Chapter 4 we described the requirements that have to be met by the extended version of ORS. In this chapter we present the mark-up we developed for the ontology protection. Our solution affects three subsystems of ORS: the Translation System, the Refinement System and the Agent Communication System. We describe the changes we made to these components and discuss the obtained results.

## 5.1 Mark-up for Ontology Protection

Each agent using ORS is equipped with an ontology and a meta-ontology. The first one provides ground-level information about the domain and the latter one: information about the ontology. Definition and information about ontology protection are hence included in the meta-ontology.

The available types of protection were thoroughly described in Section 4.5. In Appendix A.3 we attach the code that should be included in the meta-ontology if the user decides to use ontology protection. As described earlier, we use the protection of:

- **a predicate**

It contains information about the predicate name, type of protection and the level of protection.

It is of the form:

```
(Protect-Predicate <NameOfPredicate> <ProtectionType>
<ProtectionLevel>),
```

where `ProtectionType` is one of the:

- `Predicate-All`
- `Predicate-Arity`

and `ProtectionLevel` takes values:

- `High-Protection`
- `Low-Protection`

- **an argument**

It contains information about predicate name, position of an argument in the predicate, type of protection, and the level of protection.

It is of the form:

```
(Protect-Argument <NameOfPredicate> <Position> <ProtectionType>
<ProtectionLevel>),
```

where `ProtectionType` is one of the:

- `Argument-All`
- `Argument-Class`

and `ProtectionLevel` takes values:

- `High-Protection`
- `Low-Protection`

For example, in order to prevent removing the quality of treatment from `treats(plantZero,x,perfect)`<sup>1</sup>, we could either protect *the whole predicate*:

```
(Protect-Predicate Treats Predicate-All High-Protection)
or
(Protect-Predicate Treats Predicate-All Low-Protection)
```

or *the predicate's arity*:

```
(Protect-Predicate Treats Predicate-Arity High-Protection)
or
(Protect-Predicate Treats Predicate-Arity Low-Protection)
```

or *the whole argument* at the position 3:

```
(Protect-Argument Treats 3 Argument-All High-Protection)
or
(Protect-Argument Treats 3 Argument-All Low-Protection)
```

An agent must store details about agents it can interact with and about itself. These definitions are known as *frames* in KIF. Facts concerning agents, including information about ontology protection, are put into *axioms*<sup>2</sup>.

---

<sup>1</sup>Definition of *treats*:

```
treats(Agent,Contaminant,Quality)
```

<sup>2</sup>An example of a definition of *plantZero* that contains information about ontology protection:

```
(Define-Frame Plant-Zero :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)
:Axioms ((Protect-Predicate Treats Predicate-Arity Low-Protection)
(Protect-Argument Treats 3 Argument-All High-Protection )
(Protect-Argument Max-Volume 2 Argument-Value High-Protection)))
```

## 5.2 Extensions to the Translation System

The Translation System is involved in two important tasks of this project, translation of:

1. information about protection
2. information specific for SPAs

In both situations, the data is included in the meta-ontology, expressed in KIF. The Prolog version is then produced to make the information accessible by other subsystems.

### 5.2.1 Translation of Protection Information

In Section 5.1 we showed the format to express the information about protection. Note that types of arguments and the arities are different for protection of a predicate or an argument. It was decided to store this information in two types of lists:

- `protectPredicate([])`
- `protectArgument([])`

Having a Prolog version of meta-ontology enables the Refinement System to access it and behave accordingly.

The information included in the Prolog representation can be accessed by the Repair System and has influence on the outcome of the repair. We have decided to use lists of predicates as opposed to single predicates in order to make ORS more general.<sup>3</sup>

Here we present an example of the mapping between the information about protection represented in KIF and the corresponding one in Prolog:

```
(Protect-Predicate Treats Predicate-Arity Low-Protection)
↳
protectPredicate(['(treats,predicateArity,lowProtection)']).
```

and

```
(Protect-Argument Max-Volume 2 Argument-Value High-Protection)
(Protect-Argument Treats 3 Argument-All High-Protection )
```

---

<sup>3</sup>Use of single predicates would require an agent to have at least one protection of each type. Otherwise, errors in Prolog would occur. In our solution, agents that do not use the protection functionality, can access the Repair System without any problems. Simply, their lists of predicates are empty.

```

↳
protectArgument(['(maxVolume,2,argumentValue,highProtection)',
'(treats,3,argumentAll,highProtection)']).

```

### 5.2.2 Translation of Information Specific for SPAs

In Section 4.6 we explained the need for SPAs having the KIF ontologies. In the original version of the system, SPAs used only the Prolog representations. It was necessary to change this situation and modify the definition of the SPA. In our solution, the translation is called by the SPA at the beginning of the interaction and after any repairs. The newest version of the ontology is then reconsulted.

In Section 4.6, we also included information specific only for the SPAs. The Translation System was extended to consider this part of the ontology. We present an example of mapping between the version in KIF and the one in Prolog, using part of the ontology of *plantOne*<sup>4</sup>:

```

(Tasks-I-Perform Send-Contaminated-Water) ↳
tasksIPerform([sendContaminatedWater]).

```

```

(Ask Treats) ↳
ask([treats]).

```

```

(Wait Calculation) ↳
wait([calculation]).

```

```

(Non-Facts Member) (Non-Facts Assert) (Non-Facts Class) ↳
nonFacts([class,assert,member]).

```

## 5.3 Extensions to the Repair System

The aim of the Repair System is to use information obtained from the Diagnostic System to perform the repair and inform ORS about the result of this action. Earlier, it was assumed that the PA always agreed to the diagnosis and the repair. In our version, the Repair System has an additional task: it consults the meta-ontology of the agent to check information about the protection. It then acts accordingly, either performing the repair or failing to do so. In the latter case, the relevant information is passed to the agent so it can be used for repair sharing.

---

<sup>4</sup>An example of a definition of *plantOne* that contains information specific for SPAs:  
(Define-Frame Plant-One :Own-Slots ((Documentation "Not supplied yet.")  
(Instance-Of Agent)) :Axioms (  
(Tasks-I-Perform Send-Contaminated-Water) (Ask Treats) (Wait Calculation)  
(Inform At) (Non-Facts Member) (Non-Facts Assert) (Non-Facts Class)))

For each repair, we have a set of associated types of protection that influence the outcome of the repair attempt. For example, in Section 5.1 we showed that, in order to perform *propositional refinement*, we should verify the protection of: the whole predicate, the predicate's arity and the whole argument at position 3.

In Section 3.1 we presented the repairs that are in the scope of ORS. Analysis of the functionality of the system was followed by the decision about the kinds of repairs that are prevented by each type of protection. We obtained the following results:

- **Protection of a predicate**
  - **whole predicate**  
*predicate refinement, propositional refinement, propositional abstraction, domain abstraction, switch arguments, precondition abstraction, postcondition refinement, postcondition negation*
  - **predicate's arity**  
*propositional refinement, propositional abstraction*
- **Protection of an argument**
  - **whole argument**  
*propositional refinement, domain abstraction, switch arguments*
  - **argument's type**  
*domain abstraction*

As discussed in Section 4.7, the information about the repair received by the SPA is based on the diagnosis obtained by the PA. Although rules for checking protection information stay the same, information about the repair and the outcome vary. We should bear in mind that the SPA is supposed to perform the inverse of the PA's repair, but having only the information the PA obtained. Therefore, it was necessary to provide separate functions for the SPAs.

## 5.4 Extensions to the Communication System

In the original version of the system, the communication between agents focused on exchanging information about tasks and verifying whether the preconditions of the actions are met. No information concerning repair sharing was ever exchanged. It was hence necessary to design a communication model to enable the PA request the SPA to perform the repair and let the SPA reply with the outcome of this action.

We decided to add two new types of queries which are of the form:

`query(pa,Agent,<QueryType>,[Scenario,InverseRepairType,RepairInfo]),`

where `InverseRepairType` is a repair that should be performed by the SPA being an inverse of the repair proposed for the PA. `RepairInfo` provides all the information about the repair, which were given to the PA by the Diagnostic Algorithm.

Consider a situation, where the precondition `at`<sup>5</sup> causes a problem in the interaction. The position of arguments of this predicate is different for both agents:

PA: `at(plantZero,x)`

SPA: `at(x,plantZero)`

The repair proposed by the Diagnostic System is hence *switch arguments* and the information about repair contains the name of the predicate, i.e. `at`<sup>6</sup>. If the PA is unable to perform the repair, it will send the request containing the following information: `InverseRepairType` - `switchArgs` and `RepairInfo` - `at`.

`QueryType` takes values:

- **repair**

This query is used in the first attempt of requesting the SPA to perform the repair. Given the information about the repair, the SPA checks whether it is possible to perform it. If the outcome is negative, it obtains the level of the protection. The reply is sent back to the PA.

- **repair2**

This query is used in the second attempt of requesting the SPA to perform the repair. This means that the part of the ontology is protected for both agents but it might be possible that for one of them the protection is *low* hence it could agree to perform the repair.

Figure 5.1 presents the sequence diagram of the system. The outcome of the negotiation depends on the level of protection of the ontologies of interacting agents. Consider a mismatch described above where the positions of arguments of the predicate `at` vary. The diagnosis is: **switch arguments**. We shows examples of the interaction using different levels of protection:

- PA: `highProtection/lowProtection`,  
SPA: `noProtection`

I am asking SPA to perform `switchArgs`  
 SPA has received a query [`water2,switchArgs,At`] `repair`  
 I was asked to repair my ontology.

---

<sup>5</sup>A definition of `at`:  
`at(Agent,Number)`

<sup>6</sup>Switching arguments is available only for the binary predicates.

# SEQUENCE

# DIAGRAM

Figure 5.1: Sequence diagram)

I must check if the repair can be performed.  
 I agree to perform the repair.  
     SPA agreed to implement the inverse.

- PA: highProtection  
 SPA: lowProtection

    I am asking SPA to perform switchArgs  
 SPA has received a query [water2,switchArgs,At] repair  
 I was asked to repair my ontology.  
 I must check if the repair can be performed.  
 This part of my ontology is protected.  
 I refuse to perform the repair.  
     SPA cannot do it  
     My protection is high.  
     I am asking SPA to try again.  
 SPA has received a query [water2,switchArgs,At] repair2  
 I was asked to repair my ontology.  
 I must check if the repair can be performed.  
 I finally agree to perform the repair  
     SPA agreed to perform the inverse.

- PA: highProtection  
 SPA: highProtection

    I am asking SPA to perform switchArgs  
 SPA has received a query [water2,switchArgs,At] repair  
 I was asked to repair my ontology.  
 I must check if the repair can be performed.  
 This part of my ontology is protected.  
 I refuse to perform the repair.  
     SPA cannot do it  
     My protection is high.  
     I am asking SPA to try again.  
 SPA has received a query [water2,switchArgs,At] repair2  
 I was asked again to repair my ontology.  
 I must check if the repair can be performed.  
 I refuse again.  
     SPA refused to implement the inverse.  
     SPA will not be able to perform this type of  
     repair for me.

- PA: lowProtection  
 SPA: highProtection/lowProtection

    I am asking SPA to perform switchArgs

SPA has received a query [water2,switchArgs,At] repair  
I was asked to repair my ontology.  
I must check if the repair can be performed.  
This part of my ontology is protected.  
I refuse to perform the repair.  
    SPA could not do it.  
    My protection is low.  
    I am performing the repair.

## 5.5 Summary

In this chapter we have discussed how the negotiation between agents was developed using information about protection. We have described what decisions have been taken to enable any agent to use ORS. We have presented effects that our design decisions had on ORS.



# 6. Evaluation

The goals presented in Section 4.1 have been achieved. We have succeeded in designing a mark-up for ontology protection and improving the communication system to enable “repair sharing” between agents. Evaluation is necessary to verify whether the modifications we implemented improve the functionality of ORS. In this chapter we describe the experiments we conducted together with the obtained results.

## 6.1 Experiments

The aim of evaluation is to examine the hypotheses mentioned in Section 4.4:

1. This technique avoids *bad plans* by protecting important parts of the ontology. The number of plans developed decreases but the new set of plans include only *good plans*.
2. This technique provides alternative *good plans* by sharing repairs between agents.

We have decided to use two scenarios:

- The Water Management Scenario - described in Section 6.2.  
It involves a group of cooperating agents, each of which is the best authority on some facts in its ontology. The good performance of the system is a mutual goal of all the agents.
- The Shopping Scenario - described in Section 6.3.  
In this scenario it is the PA that is particularly interested in achieving the goal. However SPAs make some effort to fulfil the customer’s needs, especially if payments were involved.

We believe that using two types of scenarios provides us with a stronger evaluation.

It is necessary to compare the performance of the system with its original version and with the changes we have presented. We conduct experiments using the:

1. Original ORS
2. ORS & ontology protection
3. ORS & ontology protection & repair sharing

## 6.2 The Water Management Scenario

This scenario is a simulation that models how waste water treatment plants store water and reduce the amount of chemicals in it. The main goal of the system is the maintainance of the best water quality. Each agent is responsible for treatment of particular kinds of contaminants. Each plant has information about the volume of water it currently stores and how much more water can still be received. The KIF ontologies of the PA for this scenario can be found in Appendix A.1.

### 6.2.1 Case A

The goal of this scenario is to reduce the amount of contaminant  $x$  in the system. The quality of the treatment must be *perfect*. Consider a situation where the PA,  $plant_0$ , is able to treat<sup>1</sup> contaminant  $x$ . According to the representation of the PA,  $plant_1$  stores contaminant  $x$  it cannot treat itself. Hence, the first task is to send water containing  $x$  to  $plant_0$ . When this action succeeds, the treatment can start. However, it has to be checked that the plant storing the contaminant can in fact treat it. As the result, we obtain water without contaminant  $x$  in it.

Goal: `goodPerformance(plantZero)`.

Plan:

1. `sendContaminatedWater(plantOne,plantZero,x,perfect)`
2. `treatContaminatedWater(plantZero,x,perfect)`

From the description of this scenario we realise that the precondition, checking whether the plant is capable of treating the contaminant, appears in two actions. Therefore, if this predicate is affected by the repair, further plan execution may be impossible. When analysing preconditions, we shall only focus on the predicate `treats`.

Appendix B.1 provides the ORS output of the simulation of the Case A.

#### 6.2.1.1 Case A1

In this case we show how the execution of the plan looks if no mismatches occur:

$plant_0$ : `treats(plantZero,x,perfect)`  
 $plant_1$ : `treats(plantZero,x,perfect)`

---

<sup>1</sup>`treats(plantZero,x,perfect)`

The PA forms the plan as stated above and assign agents for the actions. *plant*<sub>1</sub> is chosen to perform `sendContaminatedWater`, as it is marked as capable of doing it and according to the PA's ontology, it stores *x*. *plant*<sub>1</sub> accepts the request to perform the task, checks the preconditions and sends the contaminated water. As a result, *x* is in *plant*<sub>0</sub> and not in *plant*<sub>1</sub>.

```
plantOne asked pa about treats(plantZero,x,perfect)
      pa has replied treats(plantZero,x,perfect)
plantOne asked pa about at(plantOne,x)
      pa has replied at(plantOne,x)
```

*plant*<sub>3</sub> is chosen to perform the second task. All the preconditions are met and the action is performed successfully.

```
plantThree asked pa about treats(plantZero,x,perfect)
      pa has replied treats(plantZero,x,perfect)
plantThree asked pa about at(plantZero,x)
      plantZero has replied at(plantZero,x)
```

The plan is completed.

### 6.2.1.2 Case A2

In this case we verify how the situation looks when the mismatch in the predicate `treats` occurs between the PA (*plant*<sub>0</sub>) and the SPA (*plant*<sub>1</sub>):

```
plant0: treats(plantZero,x,perfect)
plant1: treats(plantZero,x)
```

According to the PA's ontology, there is no mismatch in `treats` between the agents. Therefore, as before, *plantOne* is chosen to perform the task. The mismatch is discovered when the preconditions are checked:

```
plantOne asked pa about treats(plantZero,x)
      pa has replied no
```

The PA calls ORS for help and receives the diagnosis: **propositional abstraction**. The PA agrees to perform the repair:

```
treats(plantZero,x,perfect) ↦ treats(plantZero,x)
```

After the replanning, *plant*<sub>1</sub> is asked again to perform `sendContaminatedWater`. In this case, the action finishes successfully:

```
plantOne asked pa about treats(plantZero,x)
      pa has replied treats(plantZero,x)
```

As in the previous case, the PA asks *plant*<sub>3</sub> to perform: `treatContaminatedWater`. The change in the ontology, however, violates the interaction:

```
plantThree asked pa about treats(plantZero,x,perfect)
      pa has replied no
```

Although the PA attempts to fix this situation, the mismatch between the ontologies is not removed and the action cannot be performed.

The plan fails.

### 6.2.1.3 Case A3

In this case we use the same mismatch as before:

```
plant0: treats(plantZero,x,perfect)
plant1: treats(plantZero,x)
```

The previous situation showed that no protection of `treats` causes problems in further interaction. In this case we use a protection of the third argument of the predicate in order to prevent losing information about the quality of treatment:

```
plant0: (treats,3,argumentAll,highProtection)
```

As before, the mismatch occurs during the interaction with *plant*<sub>1</sub>:

```
plantOne asked pa about treats(plantZero,x)
      pa has replied no
```

The diagnosis proposed by ORS is again: **propositional abstraction**. The difference in this case is that the PA verifies whether it is possible to change the problematic part of the ontology.

```
I must check if the repair can be performed.
Protection used: (treats,3,argumentAll,highProtection)
Sorry, the repair cannot be performed
```

The PA is unable to perform the repair, hence it asks the SPA to do so. As mentioned earlier, the SPA is unable to perform the inverse of propositional abstraction, i.e. propositional refinement. Therefore it behaves as if this part of its ontology was highly protected:

```
I am asking SPA to perform propositionalAA
plantOne has received a query [water2,propositionalAA,[Treats,3]]
repair
I was asked to repair my ontology.
I must check if the repair can be performed.
This part of my ontology is protected.
```

I refuse to perform the repair.  
     plantOne cannot do it  
     My protection is high.  
     I am asking SPA to try again.  
 plantOne has received a query [water2,propositionalAA,[Treats,3]]  
 repair2  
 I was asked again to repair my ontology.  
 I must check if the repair can be performed.  
 I refuse again.  
     This part of my ontology is protected.  
     plantOne refused to implement the inverse.  
     I asked twice.  
     plantOne will not be able to perform this type of repair  
     for me.

When the relevant part of the ontology is highly protected in both the PA and the SPA's ontology and none of them is able to repair, the only solution is to find another agent to perform the action. In our scenario, there are two agents that are capable of performing the first action, i.e. `plantOne` and `plantFour`. Therefore `plantFour` is asked to perform `sendContaminatedWater`. This interaction succeeds:

```
plantFour asked pa about treats(plantZero,x,perfect)
    pa has replied treats(plantZero,x,perfect)

treatContaminatedWater is successfully performed (as in Case 0).

The plan is completed.
```

### 6.2.2 Case B

The goal of this scenario is to supply the PA with chlorine so that the treatment can be performed. Some preconditions have to be met to enable the execution of this action. A level of water in the plant must exceed the critical volume for the treatment and the plant has to currently store the contaminant it is able to treat. According to the representation of the PA, contaminant  $x$  that is treated by the PA is  $at^2$  the PA. The current level of water is below the critical value. Therefore, it is necessary to provide  $plant_0$  with water. Once this action is performed, the chlorine can be sent to the PA. As the result, we obtain the PA with chlorine. The precondition `at`, verifying whether the plant stores the contaminant that it is supposed to treat, is checked twice, before each action of the plan.

Goal: `withChlorine(plantZero)`

---

<sup>2</sup>`at(plantZero,x)`

Plan:

1. `sendWater(plantZero,x)`
2. `sendChlorine(plantZero,x)`

In this scenario we examine the repair sharing using the mismatch in the predicate `at`. The position of the arguments will be different for the PA and the SPA.

When analysing preconditions, we shall only focus on the predicate `at` as it is the one that disturbs the interaction.

Appendix B.2 provides the ORS output of the simulation of the Case B.

### 6.2.2.1 Case B1

In the case B we shall use a mismatch:

```
plant0: at(plantZero,x)
plant2 at(x,plantZero)
```

The protection used by the agents is:

```
plant0: (at,predicateAll,highProtection)
plant2: (at,predicateAll,highProtection)
```

During the execution of the first action, the following mismatch occurs:

```
plantTwo asked pa about at(x,plantZero)
      pa has replied no
```

The action cannot be performed and the diagnosis proposed by ORS is: **switch arguments** Both of the agents use high protection over the predicate `at`, therefore none of them can perform the repair. The negotiation is analogous to the one illustrated in Case A3.

Fortunately, another agent `plant6` can also able to perform `sendWater`.

```
plantSix asked pa about at(plantZero,x)
      pa has replied at(plantZero,x)
```

No mismatch occurs during the execution of the second task, hence it is performed successfully.

The plan is completed.

### 6.2.2.2 Case B2

As before, we have:

*plant*<sub>0</sub>: at(plantZero,x)  
*plant*<sub>2</sub>: at(x,plantZero)

In this case we show, that low protection of the PA's ontology protects the information only if the SPA has no protection at all:<sup>3</sup>

*plant*<sub>0</sub>: (at,predicateAll,lowProtection)

The mismatch occurs during the interaction between the PA and *plant*<sub>2</sub>. The diagnosis is: **switch arguments**. The PA cannot perform the repair therefore it asks the SPA to do it instead. The SPA agrees to modify the ontology.

I am asking SPA to perform switchArgs  
 plantTwo has received a query [water2,switchArgs,At] repair  
 I was asked to repair my ontology.  
 I must check if the repair can be performed.  
 I agree to perform the repair.  
     plantTwo agreed to implement the inverse.

### 6.2.2.3 Case B3

As before, we have:

*plant*<sub>0</sub>: at(plantZero,x)  
*plant*<sub>2</sub>: at(x,plantZero)

The protection used in this case is:

*plant*<sub>0</sub>: (at,predicateAll,highProtection)  
*plant*<sub>2</sub>: (at,predicateAll,lowProtection)

The mismatch occurs during the interaction between the PA and *plant*<sub>2</sub>. The diagnosis is: **switch arguments**. The PA cannot perform the repair. As the ontology of the SPA is protected, it refuses to the first repair request. However, level of protection is low, hence the second reply is positive.

I am asking SPA to perform switchArgs  
 plantTwo has received a query [water2,switchArgs,At] repair  
 I was asked to repair my ontology.  
 I must check if the repair can be performed.  
 This part of my ontology is protected.  
 I refuse to perform the repair.  
     My protection is high.  
     I am asking SPA to try again.

---

<sup>3</sup>Note that we always check first whether the PA is able to perform the repair. If both agents have the level of protection, it is be PA that modifies the ontology.

```

plantTwo has received a query [water2,switchArgs,At] repair2
I was asked to repair my ontology.
I must check if the repair can be performed.
I finally agree to perform the repair
      plantTwo agreed to perform the inverse.

```

After the repair, this and the following action are successfully performed.

The plan is completed.

### 6.3 The Shopping Ontology

The Shopping Scenario is an example of an online bookstore. It is based on the scenario created for evaluation of the original version of ORS. It was necessary to write the KIF ontologies for all the SPAs because only the PA was provided with them initially. The Shopping Scenario involves different types of SPAs that are responsible for various actions, i.e. registering membership, selecting the book, adding it to the basket. The goal is to purchase the book “Our Mutual Friend”.

Goal: `has(shoppingAgent,ourMutualFriend)`

Plan:

- `joinGroup(shoppingAgent,bookShopGroup)`
- `putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)`
- `buy(shoppingAgent,ourMutualFriend)`

The KIF ontologies of the PA for this scenario can be found in Appendix A.2.

#### 6.3.1 Case C

In the Cases A and B the functionality of the ontology protection and the repair sharing were examined. In this section we aim at verifying what other repairs can be performed by the SPA.

We introduce mismatches that violate the execution of two actions of the plan:

- `putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)`  
 In order to put something to the basket, the product must be first chosen. Therefore `chooseThing`<sup>4</sup> is one of the preconditions of this action. The mismatch that occurs during the execution of this action is:

---

<sup>4</sup>`chooseThing(Agent,Item)`

```
shoppingAgent: chooseThing(shoppingAgent,ourMutualFriend)
putInBasketAgent: chooseItem(shoppingAgent,ourMutualFriend)
```

- buy(shoppingAgent,ourMutualFriend)

Purchase of the book can only happen when the PA proves to have enough money for it. Information about the current amount of money an agent possesses is represented by the predicate `money`<sup>5</sup>. We analyze a situation where the arity of this predicate varies and examine performance of the system when dealing with the occurring mismatches.

The mismatch that occurs during the execution of this action is:

```
shoppingAgent: money(shoppingAgent,100)
buyAgentOne: money(shoppingAgent,dollars,100)
```

Appendix B.3 provides the ORS output of the simulation of the Case B.

### 6.3.1.1 Case C1

The protection used in this case is:

```
shoppingAgent: (chooseThing,predicateAll,highProtection)
& (money,predicateArity,highProtection)
putInBasketAgentOne: (chooseItem,predicateAll,highProtection)
buyAgentOne: (money,predicateAll,highProtection)
```

In this case the mismatch occurs when the `buyAgentOne` checks the precondition `money`. The arity of the predicate is different for the PA and the SPA. The SPA has an additional argument: `dollars`. The diagnosis suggested by ORS is: **predicate anti-abstraction**, which means that the PA is supposed to add the missing argument to the predicate `money`:

```
money(shoppingAgent,100) ↦ money(shoppingAgent,dollars,100)
```

The level of protection of this part of the ontology is *high* in both cases, hence none of the interacting agents can perform the repair. It is required to find another agent to perform the task instead. The interaction with the `buyAgentTwo` is successful.

Another of the mismatches introduced in this case is detected during the interaction of the PA with `putInBasketAgentOne`. The name of one of the preconditions is different, but arguments stay the same. The repair suggested by ORS is: **propositional anti-abstraction**. The name of the precondition should be then changed:

```
chooseThing(shoppingAgent,ourMutualFriend) ↦
```

---

<sup>5</sup>`money(Agent,Amount)`

```
chooseItem(shoppingAgent,ourMutualFriend)
```

Also in this situation it is necessary to find another agent, *putInBasketAgentTwo*, to perform the action as the interaction between the PA and the SPA is impossible due to the mismatch.

### 6.3.1.2 Case C2

The protection used in this case is:

```
shoppingAgent: (chooseThing,predicateAll,highProtection)
& (money,predicateArity,highProtection)
```

In this situation the mismatches are the same as earlier. The difference is that none of the SPAs protects the ontology. The repair proposed for the PA during interaction with *buyAgentOne* was **propositional anti-abstraction**. The inverse of it, **propositional abstraction**, was successfully done by the SPA:

```
money(shoppingAgent,dollars,100)  $\mapsto$  money(shoppingAgent,100)
```

In this case also *putInBasketAgentOne* performs the repair. The mismatch concerns the name of one of the predicates. In order to solve the problem, the PA would perform **propositional anti-abstraction**. As this predicate is not protected in the SPA's ontology, the inverse of the PA's repair, **propositional abstraction**, is performed by *putInBasketAgentOne*:

```
chooseItem(shoppingAgent,ourMutualFriend)  $\mapsto$ 
chooseThing(shoppingAgent,ourMutualFriend)
```

The plan is completed.

## 6.4 Results

The aim of the evaluation was to examine whether the ontology protection and repair sharing improve the functionality of the system. The positive results of the evaluation support the initial hypotheses of this project.

The Case A verified the importance of ontology protection. We showed that it is important to use ontology protection for the information that is necessary in the future. Otherwise, this information might be lost what results in the failure of the plan.

Cases A3, B1 and C1 showed that although ontology protection prevents the execution of an action, it is possible to find alternative solutions to achieve the goal. When none of the interacting agents can perform the repair, the PA tries

to find another agent to perform the task instead. We believe that it is worth paying the cost of finding an alternative solution but keeping the protected parts of the ontology secure.

In the Case B we used different levels of protection to examine the functionality of the negotiation about repairs. In all the cases the results were as expected.

In the evaluation we verified different types of mismatches, repairs and protection. Our solution worked in all of these cases. However, there are more mismatches that are solved by the extended version of the system. In order to fully evaluate the system it would be necessary to design new scenarios and introduce these type of mismatches that we have not examined.

## 6.5 Summary

In this Chapter we have examined the functionalities of the extended version of the system. We have conducted experiments using different types and levels of protection.



# 7. Further Work

## 7.1 Richer diagnosis

Our approach enables SPAs to perform abstraction of the parts of the ontology or switch arguments of the predicates. Refinement, however, is not accessible for SPAs. This comes from the fact that all the information about the repair of the SPA is based only on the information that the PA received from ORS for itself. This does not cause problems when the SPA is supposed to switch arguments or perform the refinement. However, abstraction requires more information, i.e. what is the type and the value of an argument to be added, what is its new position. When the PA asks the Diagnostic Algorithm for the diagnosis, it only receives the information about the repair that it should perform. No information about possible repair for the SPA is included. Having all the information to repair any of the two ontologies is necessary for the negotiation. In order to obtain the richer diagnosis, it is necessary to make changes to the Diagnostic Algorithm.

## 7.2 Complex mismatches

In the current version of the system, the SPA has only access to the Translation System and the Refinement System. Letting the SPA consult the Diagnostic System would enable any agent to find the possible repair. ORS is only capable of repairing single mismatches, i.e. where the difference in the arity is equal 1 or when there is a problem with a single argument at a time. By complex mismatches, we mean a composition of single ones. ORS is able to provide solution if one of the agents has a binary predicate and the other one has a ternary one. However, mismatch between the binary and 4-ary predicates is out of the scope of the system. However, we believe that sharing repairs might be the solution to the problem. If each of the agent has access to the Diagnostic Algorithm, it might examine the problem and repair its ontology in order to minimise the mismatch between the ontologies. However, agents have no guarantee that there exists a solution leading to the success. The negotiation, as to what extent each of the agents should repair its ontology, is hence required.

### 7.3 Dependencies between arguments

Our solution provides protection of a single argument or of a predicate. However, some arguments depend on each other (e.g. a value and a measure) and if one is changed the other should be changed accordingly. This requires some kind of translation between the units. Suppose we have a predicate `money(agent, dollars, 150)`. If we want to change dollars to pounds, we need to change the value as well and hence obtain `money(agent, pounds, 100)`. Otherwise, the representation of the environment is no longer valid. All these functionalities require making changes to the Refinement System as well.

### 7.4 Broader evaluation

Complete evaluation of the solution we suggest requires more experiments. It would be advantageous to examine the functionality of the system when using different types of mismatches. However, writing new ontologies and finding appropriate types of mismatches requires a lot of time.

## 8. Conclusions

In previous chapters we observed how ontological mismatches violate the communication between agents and hence prevent execution of the plans. In Chapter 3 we presented ORS as the solution to this problem.

We suggested possible improvements of the functionality of the system. We focused on the extension of ORS that provides agents with the possibility to protect parts of the ontology. We showed that repair sharing is an advantageous solution in the agent communication.

Our solution extended the scope of ORS. The system can be used by any agent having KIF ontologies.

The results of the experiments we have conducted supported the hypotheses of this project:

1. This technique avoids *bad plans* by protecting important parts of the ontology. The number of plans developed decreases but the new set of plans include only *good plans*.
2. This technique provides alternative *good plans* by sharing repairs between agents.



# Bibliography

- [1] *KIF user's manual*.  
<http://www-ksl.stanford.edu/knowledge-sharing/kif/>, March 2011.
- [2] *Ontolingua Reference Manual*.  
<http://www-ksl.stanford.edu/htw/dme/thermal-kb-tour/ONTOLINGUA.html>, March 2011.
- [3] *Sicstus user's manual*.  
<http://www.sics.se/sicstus/docs/latest/html/sicstus/>, March 2011.
- [4] J. L. Austin. *How to Do Things with Words*. Harvard University Press, 1962.
- [5] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web: Scientific American. *Scientific American*, 2001.
- [6] A. Bundy and F. McNeill. Representation as a fluent: An AI challenge for the next half century, 2006.
- [7] T. R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.
- [8] F. McNeill. *Dynamic Ontology Refinement*. PhD thesis, Division of Informatics, University of Edinburgh.
- [9] F. McNeill and A. Bundy. Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution. *IJSWIS (International Journal on Semantic Web and Information Systems) special issue on Ontology Matching*, 3:1–35, 2007.
- [10] S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.
- [11] G. Weiss. *Multiagent Systems: A modern Approach to Distributed Artificial Intelligence*. MIT Press., 1999.
- [12] M. Wooldridge. *Introduction to Multiagent Systems*. John Wiley Systems and Sons, 2002.



# Appendix A. Ontologies

In this section we include ontologies of the scenarios described in Chapter 6.

## A.1 The Water Management Scenario Ontologies

### A.1.1 ont.in

```
;;; Agent (Define-Class Agent (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Good-Performance
(Define-Relation Good-Performance (?Agent ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Sit-Var ?Situation)))

;;; With-Chloride
(Define-Relation With-Chloride (?Agent ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Sit-Var ?Situation)))

;;; Enough-Water
(Define-Relation Enough-Water (?Agent ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Sit-Var ?Situation)))

;;; Contaminant
(Define-Class Contaminant (?X) "Not supplied yet." :Def (And (Thing
?X)))

;;; Quality
(Define-Class Quality (?X) "Not supplied yet." :Def (And (Thing
?X)))

;;; Sent
(Define-Class Sent (?X) "Not supplied yet." :Def (And (Thing
(?X))))

;;; Critical-Volume
(Define-Function Critical-Volume (?Agent ?Situation) :-> ?Value "Not
```

```

supplied yet." :Def (And (Agent ?Agent) (Number ?Value) (Sit-Var
?Situation)))

;;; Used-Volume
(Define-Function Used-Volume (?Agent ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Agent ?Agent) (Number ?Value) (Sit-Var
?Situation)))

;;; Free-Volume
(Define-Relation Free-Volume (?Agent ?Value ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Number ?Value) (Sit-Var ?Situation)))

;;; Treats
(Define-Relation Treats (?Agent ?Contaminant ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Contaminant ?Contaminant) (Quality
?Quality) (Sit-Var ?Situation)))

;;; At
(Define-Relation At (?Agent ?Contaminant ?Situation) "Not supplied yet."
:Def (And (Agent ?Agent) (Contaminant ?Contaminant) (Sit-Var ?Situation)))

;;; Plant-Zero
(Define-Frame Plant-Zero :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((At Plant-Zero X [(Start)]) (Used-Volume
Plant-Zero 100 [(Start)]) (Free-Volume Plant-Zero 400 [(Start)])
(Critical-Volume Plant-Zero 200 [(Start)])))

;;; Plant-One
(Define-Frame Plant-One :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((At Plant-One X [(Start)])
(Free-Volume Plant-One 400 [(Start)])))

;;; Plant-Two
(Define-Frame Plant-Two :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Used-Volume Plant-Two 200 [(Start)])
(Free-Volume Plant-Two 100 [(Start)])))

;;; Plant-Three
(Define-Frame Plant-One :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((At Plant-Three Beta [(Start)]) (Free-Volume
Plant-Three 400 [(Start)])))

```

```

;;; Plant-Four
(Define-Frame Plant-One :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((At Plant-Four X [(Start)])))

;;; Send-Contaminated-Water
(Define-Axiom Send-Contaminated-Water "Not supplied yet." := (=> (And
(At ?Agent1 ?Contaminant ?Sit1) (Treats ?Agent2 ?Contaminant ?Quality
?Sit1)) (And (At ?Agent2 ?Contaminant ?Sit2) (Not (At ?Agent1
?Contaminant ?Sit2)))))

;;; Treat-Contaminated-Water
(Define-Axiom Treat-Contaminated-Water "Not supplied yet." := (=> (And
(At ?Agent1 ?Contaminant ?Sit1) (Quality ?Quality) (Treats ?Agent1
?Contaminant ?Quality ?Sit1)) (And (Good-Performance ?Agent1 ?Sit2))))

;;; Send-Water
(Define-Axiom Send-Water "Not supplied yet." := (=> (And (Used-Volume
?Agent1 ?Used1 ?Sit1) (Critical-Volume ?Agent1 ?Crit1 ?Sit1) (At ?Agent1
?Contaminant) (< ?Used1 ?Crit1)) (Contaminant ?Contaminant)) (And
(Enough-Water ?Agent1 ?Sit2))))

;;; Send-Chloride
(Define-Axiom Send-Chloride "Not supplied yet." := (=> (And (Enough-Water
?Agent1 ?Sit1) (At ?Agent1 ?Contaminant)) (And (With-Chloride
?Agent1))))

;;; Perfect
(Define-Individual Perfect (Quality) "Not supplied yet.")

;;; X
(Define-Individual X (Contaminant) "Not supplied yet.")

;;; Y
(Define-Individual Y (Contaminant) "Not supplied yet.")

;;; Sit-Var
(Define-Class Sit-Var (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Start
(Define-Individual Start (Sit-Var) "Not supplied yet.")

```

### A.1.2 metaOnt.in

```
;;; Action (Define-Class Action (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Inform
(Define-Class Inform (?X) "Not supplied yet." :Def (And (Predicate ?X)))

;;; Agent-Needed
(Define-Function Agent-Needed (?Agent-0) :-> ?Value "Not supplied yet."
:Def (And (Agent ?Agent-0) (Action ?Value)))

;;; Plant-One
(Define-Frame Plant-One :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Plant-One
Send-Contaminated-Water)))

;;; Plant-Two
(Define-Frame Plant-Two :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Plant-Two Send-Water)))

;;; Plant-Three
(Define-Frame Plant-Three :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Plant-Three
Treat-Contaminated-Water)))

;;; Plant-Four
(Define-Frame Plant-Four :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Plant-Four
Send-Contaminated-Water)))

;;; Plant-Five
(Define-Frame Plant-Five :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Plant-Five Send-Chloride)))

;;; Plant-Six
(Define-Frame Plant-Six :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed Plant-Six
Send-Water)))
```

```

;;; Plant-Zero
(Define-Frame Plant-Zero :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Agent)) :Axioms ((Wait Calculation)
(Protect-Predicate At Predicate-All High-Protection) (Protect-Argument
Treats 3
Argument-All High-Protection)))

;;; Send-Contaminated-Water
(Define-Individual Send-Contaminated-Water (Action) "Not supplied
yet.")

;;; Treat-Contaminated-Water
(Define-Individual Treat-Contaminated-Water (Action) "Not supplied
yet.")

;;; Send-Water
(Define-Individual Send-Water (Action) "Not supplied yet.")

;;; Send-Chlorine
(Define-Individual Send-Chlorine (Action) "Not supplied yet.")

;;; PROTECTED

;;; Predicate-Option
(Define-Class Predicate-Option (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Argument-Option
(Define-Class Argument-Option (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Protection-Level
(Define-Class Protection-Level (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Protect-Predicate
(Define-Function Protect-Predicate (?Relation ?Value) :-> ?Level
"Not supplied yet." :Def (And (Relation ?Relation) (Predicate-Option
?Value) (Protection-Level ?Level)))

;;; Protect-Argument
(Define-Function Protect-Argument (?Relation ?Argument ?Value) :-> ?Level

```

```

"Not supplied yet." :Def (And (Relation ?Relation) (Argument ?Argument)
(Argument-Option ?Value) (Protection-Level ?Level)))

;;; Predicate-All
(Define-Individual Predicate-All (Predicate-Option) "Not supplied yet.")

;;; Predicate-Arity
(Define-Individual Predicate-Arity (Predicate-Option) "Not supplied
yet.")

;;; Argument-All
(Define-Individual Argument-All (Argument-Option) "Not supplied yet.")

;;; Argument-Value
(Define-Individual Argument-Value (Argument-Option) "Not supplied yet.")

;;; Argument-Class
(Define-Individual Argument-Class (Argument-Option) "Not supplied yet.")

;;; High-Protection
(Define-Individual High-Protection (Protection-Level) "Not supplied
yet.")

;;; Low-Protection
(Define-Individual Low-Protection (Protection-Level) "Not supplied yet.")

;;; PROTECTED

```

## A.2 The Shopping Scenario Ontologies

### A.2.1 ont.in

```

;;; Item
(Define-Class Item (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Group

```

```

(Define-Class Group (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Shopping-Group
(Define-Class Shopping-Group (?X) "Not supplied yet." :Def (And (Group
?X)))

;;; Academic-Group
(Define-Class Academic-Group (?X) "Not supplied yet." :Def (And (Group
?X)))

;;; Agent
(Define-Class Agent (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Book
(Define-Class Book (?X) "Not supplied yet." :Def (And (Item ?X)))

;;; Buy
(Define-Axiom Buy "Not supplied yet." := (=> (And (In-Basket ?Agent
?Item Pseudo-Var ?Sit1) (Money ?Agent ?Amount ?Sit1) (Cost ?Item ?Price
?Sit1) (< ?Price ?Amount)) (And (Has ?Agent ?Item ?Sit2) (= ?Newamount
(- ?Amount ?Price)) (Money ?Agent ?Newamount ?Sit2) (Not (Money ?Agent
?Amount ?Sit2))))))

;;; Has
(Define-Relation Has (?Agent ?Item ?Situation) "Not supplied yet." :Def
(And (Agent ?Agent) (Item ?Item) (Sit-Var ?Situation)))

;;; In-Basket
(Define-Function In-Basket (?Agent-0 ?Item ?Situation) :-> ?Value "Not
supplied yet." :Def (And (Agent ?Agent-0) (Item ?Item)
(Confirmation-Number ?Value) (Sit-Var ?Situation)))

;;; Choose-Thing
(Define-Relation Choose-Thing (?Agent ?Item ?Situation) "Not supplied
yet." :Def (And (Agent ?Agent) (Item ?Item) (Sit-Var ?Situation)))

;;; Registered-Member
(Define-Relation Registered-Member (?Agent ?Group ?Situation) "Not
supplied yet." :Def (And (Agent ?Agent) (Group ?Group) (Sit-Var
?Situation)))

```

```

;;; Cost
(Define-Function Cost (?Item ?Situation) :-> ?Value "Not supplied yet."
:Def (And (Item ?Item) (Number ?Value) (Sit-Var ?Situation)))

;;; Money
(Define-Function Money (?Agent-0 ?Situation) :-> ?Value "Not supplied
yet." :Def (And (Agent ?Agent-0) (Number ?Value) (Sit-Var ?Situation)))

;;; Put-In-Basket
(Define-Axiom Put-In-Basket "Not supplied yet." := (=> (And (Choose-Thing
?Agent ?Item ?Sit1) (Registered-Member ?Agent ?Group) (Group ?Group))
(And (In-Basket ?Agent ?Item Pseudo-Var ?Sit2))))

;;; Join-Group
(Define-Axiom Join-Group "Not supplied yet." := (=> (And (Group ?Group))
(And (Registered-Member ?Agent ?Group))))

;;; Our-Mutual-Friend
(Define-Frame Our-Mutual-Friend :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Book)) :Axioms ((Cost Our-Mutual-Friend 9
[(Start)]))

;;; Book-Shop-Group
(Define-Individual Book-Shop-Group (Shopping-Group) "Not supplied yet.")

;;; Ai-Group
(Define-Individual Ai-Group (Academic-Group) "Not supplied yet.")

;;; Shopping-Agent
(Define-Frame Shopping-Agent :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Agent) (Registered-Member Ai-Group [(Start)])
(Choose-Thing Our-Mutual-Friend [(Start)])) :Axioms ((Money
Shopping-Agent 100 [(Start)]))

;;; Currency
(Define-Class Currency (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Dollars
(Define-Individual Dollars (Currency) "Not supplied yet.")

```

```

;;; Sterling
(Define-Function Sterling (?Agent-0 ?Situation) :-> ?Value "Not supplied
yet." :Def (And (Agent ?Agent-0) (Number ?Value) (Sit-Var ?Situation)))

;;; Confirmation-Number
(Define-Class Confirmation-Number (?X) "Not supplied yet." :Def (And
(Thing ?X)))

;;; Pseudo-Var
(Define-Individual Pseudo-Var (Confirmation-Number) "Not supplied yet.")

;;; Sit-Var
(Define-Class Sit-Var (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Start
(Define-Individual Start (Sit-Var) "Not supplied yet.")

```

### A.2.2 metaOnt.in

```

;;; Buy-Agent-One
(Define-Frame Buy-Agent-One :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Agent)) :Axioms ((Agent-Needed Buy-Agent-One Buy)))

;;; Buy-Agent-Two
(Define-Frame Buy-Agent-Two :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed Buy-Agent-Two Buy)))

;;; Join-Group-Agent-One
(Define-Frame Join-Group-Agent-One :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed
Join-Group-Agent-One Join-Group)))

;;; Join-Group-Agent-Two
(Define-Frame Join-Group-Agent-Two :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed
Join-Group-Agent-Two Join-Group)))

;;; Put-In-Basket-Agent-One

```

```

(Define-Frame Put-In-Basket-Agent-One :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed
Put-In-Basket-Agent-One Put-In-Basket)))

;;; Put-In-Basket-Agent-Two
(Define-Frame Put-In-Basket-Agent-Two :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed
Put-In-Basket-Agent-Two Put-In-Basket)))

;;; Put-Item-In-Basket-Agent
(Define-Frame Put-Item-In-Basket-Agent :Own-Slots ((Documentation "Not
supplied yet.") (Instance-Of Agent)) :Axioms ((Agent-Needed
Put-Item-In-Basket-Agent Put-Item-In-Basket)))

;;; Shopping-Agent
(Define-Frame Shopping-Agent :Own-Slots ((Documentation "Not supplied
yet.") (Instance-Of Agent)) :Axioms ((Protect-Predicate Money
Predicate-Arity High-Protection) (Protect-Predicate Choose-Thing
Predicate-All High-Protection)))

;;; Choose-Item
(Define-Class Choose-Item (?X) "Not supplied yet." :Def (And
(Choose-Thing ?X)))

;;; Choose-Thing
(Define-Class Choose-Thing (?X) "Not supplied yet." :Def (And
(Predicate ?X)))

;;; Registered-Member
(Define-Individual Registered-Member (Predicate) "Not supplied yet.")

;;; In-Basket
(Define-Frame In-Basket :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of My-Fact Predicate)) :Axioms ((Inform In-Basket)))

;;; Has
(Define-Frame Has :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Predicate)) :Axioms ((My-Fact Has)))

;;; Cost
(Define-Individual Cost (Predicate) "Not supplied yet.")

```

```
;;; Money
(Define-Frame Money :Own-Slots ((Documentation "Not supplied yet.")
(Instance-Of Predicate)) :Axioms ((My-Fact Money)))

;;; Put-In-Basket
(Define-Individual Put-In-Basket (Action) "Not supplied yet.")

;;; Buy
(Define-Individual Buy (Action) "Not supplied yet.")

;;; Join-Group
(Define-Individual Join-Group (Action) "Not supplied yet.")

;;; Wait-Fact
(Define-Class Wait-Fact (?X) "Not supplied yet." :Def (And (Predicate
?X)))

;;; Predicate
(Define-Class Predicate (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Agent-Needed
(Define-Function Agent-Needed (?Agent-0) :-> ?Value "Not supplied yet."
:Def (And (Agent ?Agent-0) (Action ?Value)))

;;; Action
(Define-Class Action (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Inform
(Define-Class Inform (?X) "Not supplied yet." :Def (And (Predicate ?X)))

;;; Agent
(Define-Class Agent (?X) "Not supplied yet." :Def (And (Thing ?X)))

;;; Ask-Fact
(Define-Class Ask-Fact (?X) "Not supplied yet." :Def (And (Predicate
?X)))

;;; PROTECTED
```

```

;;; Predicate-Option
(Define-Class Predicate-Option (?X) "Not supplied yet." :Def (And (Thing
?X)))

;;; Argument-Option
(Define-Class Argument-Option (?X) "Not supplied yet." :Def (And (Thing
?X)))

;;; Protection-Level
(Define-Class Protection-Level (?X) "Not supplied yet." :Def (And (Thing
?X)))

;;; Protect-Predicate
(Define-Function Protect-Predicate (?Relation ?Value) :-> ?Level "Not
supplied yet." :Def (And (Relation ?Relation) (Predicate-Option ?Value)
(Protection-Level ?Level)))

;;; Protect-Argument
(Define-Function Protect-Argument (?Relation ?Argument ?Value) :-> ?Level
"Not supplied yet." :Def (And (Relation ?Relation) (Argument ?Argument)
(Argument-Option ?Value) (Protection-Level ?Level)))

;;; Predicate-All
(Define-Individual Predicate-All (Predicate-Option) "Not supplied yet.")

;;; Predicate-Arity
(Define-Individual Predicate-Arity (Predicate-Option) "Not supplied
yet.")

;;; Argument-All
(Define-Individual Argument-All (Argument-Option) "Not supplied yet.")

;;; Argument-Class
(Define-Individual Argument-Class (Argument-Option) "Not supplied yet.")

;;; High-Protection
(Define-Individual High-Protection (Protection-Level) "Not supplied
yet.")

;;; Low-Protection

```

```
(Define-Individual Low-Protection (Protection-Level) "Not supplied yet.")
```

```
;;; PROTECTED
```

### A.3 Mark-up for ontology protection

```
;;; Predicate-Option
```

```
(Define-Class Predicate-Option (?X) "Not supplied yet." :Def (And (Thing ?X)))
```

```
;;; Argument-Option
```

```
(Define-Class Argument-Option (?X) "Not supplied yet." :Def (And (Thing ?X)))
```

```
;;; Protection-Level
```

```
(Define-Class Protection-Level (?X) "Not supplied yet." :Def (And (Thing ?X)))
```

```
;;; Protect-Predicate
```

```
(Define-Function Protect-Predicate (?Relation ?Value) :-> ?Level "Not supplied yet." :Def (And (Relation ?Relation) (Function-Option ?Value) (Protection-Level ?Level)))
```

```
;;; Protect-Argument
```

```
(Define-Function Protect-Argument (?Relation ?Argument ?Value) :-> ?Level "Not supplied yet." :Def (And (Relation ?Relation) (Argument ?Argument) (Argument-Option ?Value) (Protection-Level ?Level)))
```

```
;;; Predicate-All
```

```
(Define-Individual Predicate-All (Function-Option) "Not supplied yet.")
```

```
;;; Predicate-Arity
```

```
Case B (Define-Individual Predicate-Arity (Function-Option) "Not supplied yet.")
```

```
;;; Argument-All
```

```
(Define-Individual Argument-All (Argument-Option) "Not supplied yet.")
```

```
;;; Argument-Class
```

```
(Define-Individual Argument-Class (Argument-Option) "Not supplied yet.")
```

```
;;; High-Protection  
(Define-Individual High-Protection (Protection-Level) "Not supplied  
yet.")
```

```
;;; Low-Protection  
(Define-Individual Low-Protection (Protection-Level) "Not supplied  
yet.")
```

# Appendix B. Evaluation outputs

## B.1 Case A

### B.1.1 Case A1

```
| ?- start.  
Consulting the PLANNER...  
Consulting the ONTOLOGY UPDATER...  
Consulting ORS...  
  
GOAL is: goodPerformance(plantZero)  
Translating ...  
Need to find a plan ...  
This is the plan:  
[sendContaminatedWater(plantOne,plantZero,x,perfect),  
treatContaminatedWater(plantZero,x,perfect)]  
Executing the plan ...
```

```
I'm going to ask plantOne to perform  
sendContaminatedWater(plantOne,plantZero,x,perfect) for me  
plantOne asked me treats(plantZero,x,perfect)  
I told plantOne treats(plantZero,x,perfect)  
plantOne asked me at(plantOne,x)  
I told plantOne at(plantOne,x)  
plantOne asked me class(plantOne,agent)  
I told plantOne class(plantOne,agent)  
plantOne asked me class(x,contaminant)  
I told plantOne class(x,contaminant)  
plantOne asked me class(plantZero,agent)  
I told plantOne class(plantZero,agent)  
plantOne asked me class(x,contaminant)  
I told plantOne class(x,contaminant)  
plantOne asked me class(perfect,quality)  
I told plantOne class(perfect,quality)  
sendContaminatedWater(plantOne,plantZero,x,perfect)  
performed successfully
```

```
I'm going to ask plantThree to perform  
treatContaminatedWater(plantZero,x,perfect) for me
```

```

plantThree asked me treats(plantZero,x,perfect)
I told plantThree treats(plantZero,x,perfect)
plantThree asked me at(plantZero,x)
I told plantThree at(plantZero,x)
plantThree asked me class(plantZero,agent)
I told plantThree class(plantZero,agent)
plantThree asked me class(x,contaminant)
I told plantThree class(x,contaminant)
plantThree asked me class(plantZero,agent)
I told plantThree class(plantZero,agent)
plantThree asked me class(x,contaminant)
I told plantThree class(x,contaminant)
plantThree asked me class(perfect,quality)
I told plantThree class(perfect,quality)
treatContaminatedWater(plantZero,x,perfect)
performed successfully
The KIF ontology has been updated
The plan is completed
The following actions have been performed:
[treatContaminatedWater(plantZero,x,perfect),
sendContaminatedWater(plantOne,plantZero,x,perfect),start]

To terminate the program, type "t."
yes

```

### B.1.2 Case A2

```

| ?- start.
Consulting the PLANNER...
Consulting the ONTOLOGY UPDATER...
Consulting ORS...
GOAL is: goodPerformance(plantZero)
Translating ...
Need to find a plan ...
This is the plan:
[sendContaminatedWater(plantOne,plantZero,x,perfect),
treatContaminatedWater(plantZero,x,perfect)] Executing the plan ...

```

```

I'm going to ask plantOne to perform
sendContaminatedWater(plantOne,plantZero,x,perfect) for me
plantOne asked me treats(plantZero,x)

```

I told plantOne no  
 The KIF ontology has been updated  
 sendContaminatedWater(plantOne,plantZero,x,perfect) failed.  
 I am asking ORS for a diagnosis.  
 I am requesting a diagnosis ...  
 I received a query about treats(plantZero,x),  
 which I was not expecting to be asked about.  
 treats(plantZero,x) has the same name as the precondition  
 treats(plantZero,x,perfect)  
 They have different arity (2 and 3)  
 DIAGNOSIS: Propositional abstraction  
 My arguments are of these types : [quality,contaminant,agent]  
 I must check if the repair can be performed.  
 I am performing the repair  
 ORS proposed the following diagnosis repair:  
 [treats(plantZero,x,perfect),treats(plantZero,x),  
 [propositionalA,[treats,3]]]  
 This repair was performed.  
 GOAL is: goodPerformance(plantZero)  
 Translating ...  
 Need to find a plan ...  
 This is the plan:  
 [sendContaminatedWater(plantOne,plantZero,x),  
 treatContaminatedWater(plantZero,x,perfect)]  
 Executing the plan ...  
 I'm going to ask plantOne to perform  
 sendContaminatedWater(plantOne,plantZero,x) for me  
 plantOne asked me treats(plantZero,x)  
 I told plantOne treats(plantZero,x)  
 plantOne asked me at(plantOne,x)  
 I told plantOne at(plantOne,x)  
 plantOne asked me class(plantOne,agent)  
 I told plantOne class(plantOne,agent)  
 plantOne asked me class(x,contaminant)  
 I told plantOne class(x,contaminant)  
 plantOne asked me class(plantZero,agent)  
 I told plantOne class(plantZero,agent)  
 plantOne asked me class(x,contaminant)  
 I told plantOne class(x,contaminant)  
 sendContaminatedWater(plantOne,plantZero,x)  
 performed successfully  
  
 I'm going to ask plantThree to perform

treatContaminatedWater(plantZero,x,perfect) for me  
plantThree asked me treats(plantZero,x,perfect)  
I told plantThree no  
The KIF ontology has been updated  
treatContaminatedWater(plantZero,x,perfect) failed.  
I am asking ORS for a diagnosis.  
I am requesting a diagnosis ...  
I received a query about treats(plantZero,x,perfect), which I was not  
expecting to be asked about.  
treats(plantZero,x,perfect) has the same name as the precondition  
treats(plantZero,x)  
They have different arity (3 and 2)  
[quality,contaminant,agent]treats requires  
an extra argument of type quality  
I already know that quality is a subclass of thing  
I must check if the repair can be performed.  
I am performing the repair  
ORS proposed the following diagnosis repair:  
propositional\_anti\_abstraction This repair was performed.  
GOAL is: goodPerformance(plantZero)  
Translating ...  
Need to find a plan ...  
This is the plan:  
[treatContaminatedWater(plantZero,x,perfect,perfect)]  
Executing the plan ...  
I'm going to ask plantThree to perform  
treatContaminatedWater(plantZero,x,perfect,perfect) for me  
The KIF ontology has been updated  
treatContaminatedWater(plantZero,x,perfect,perfect) failed.  
I am asking ORS for a diagnosis.  
I am requesting a diagnosis ...This plan failed immediately after a  
request was made to perform  
treatContaminatedWater(plantZero,x,perfect,perfect)  
plantThree says that he can perform this task  
DIAGNOSIS: problem precond: at(plantZero,x)  
at(plantZero,x) is an original fact in my ontology  
I must check if the repair can be performed.  
I am performing the repair

### B.1.3 Case A3

| ?- start.

Consulting the PLANNER...

Consulting the ONTOLOGY UPDATER...

Consulting ORS...

GOAL is: goodPerformance(plantZero)

Translating ...

Need to find a plan ...

This is the plan:

[sendContaminatedWater(plantOne,plantZero,x,perfect),

treatContaminatedWater(plantZero,x,perfect)]

Executing the plan ...

I'm going to ask plantOne to perform

sendContaminatedWater(plantOne,plantZero,x,perfect) for me

plantOne asked me treats(plantZero,x)

I told plantOne no

The KIF ontology has been updated

sendContaminatedWater(plantOne,plantZero,x,perfect) failed.

I am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about

treats(plantZero,x), which I was not expecting to be asked about.

treats(plantZero,x) has the same name as the precondition

treats(plantZero,x,perfect)

They have different arity (2 and 3)

DIAGNOSIS: Propositional abstraction

My arguments are of these types : [quality,contaminant,agent]

I must check if the repair can be performed.

Protection used: (treats,3,argumentAll,highProtection)

Sorry, the repair cannot be performed

ORS proposed the following diagnosis repair:

[treats(plantZero,x,perfect),treats(plantZero,x),[propositionalA,

[treats,3]]]

I am asking SPA to perform propositionalAA

plantOne cannot do it

My protection is high.

I am asking SPA to try again.

This part of my ontology is protected.

plantOne refused to implement the inverse.

I asked twice.

plantOne will not be able to perform this type of repair for me.  
GOAL is: goodPerformance(plantZero)  
Translating ...  
Need to find a plan ...  
This is the plan:  
[sendContaminatedWater(plantOne,plantZero,x,perfect),  
treatContaminatedWater(plantZero,x,perfect)]  
Executing the plan ...  
I'm going to ask plantFour to perform  
sendContaminatedWater(plantOne,plantZero,x,perfect) for me  
plantFour asked me treats(plantZero,x,perfect)  
I told plantFour treats(plantZero,x,perfect)  
plantFour asked me at(plantOne,x)  
I told plantFour at(plantOne,x)  
plantFour asked me class(plantOne,agent)  
I told plantFour class(plantOne,agent)  
plantFour asked me class(x,contaminant)  
I told plantFour class(x,contaminant)  
plantFour asked me class(plantZero,agent)  
I told plantFour class(plantZero,agent)  
plantFour asked me class(x,contaminant)  
I told plantFour class(x,contaminant)  
plantFour asked me class(perfect,quality)  
I told plantFour class(perfect,quality)  
sendContaminatedWater(plantOne,plantZero,x,perfect)  
performed successfully  
I'm going to ask plantThree to perform  
treatContaminatedWater(plantZero,x,perfect) for me  
plantThree asked me treats(plantZero,x,perfect)  
I told plantThree treats(plantZero,x,perfect)  
plantThree asked me at(plantZero,x)  
I told plantThree at(plantZero,x)  
plantThree asked me class(plantZero,agent)  
I told plantThree class(plantZero,agent)  
plantThree asked me class(x,contaminant)  
I told plantThree class(x,contaminant)  
plantThree asked me class(plantZero,agent)  
I told plantThree class(plantZero,agent)  
plantThree asked me class(x,contaminant)  
I told plantThree class(x,contaminant)  
plantThree asked me class(perfect,quality)  
I told plantThree class(perfect,quality)  
treatContaminatedWater(plantZero,x,perfect)

performed successfully  
 The plan is completed  
 The following actions have been performed:  
 [treatContaminatedWater(plantZero,x,perfect),  
 sendContaminatedWater(plantOne,plantZero,x,perfect),start]  
 To terminate the program, type "t."  
 yes

## B.2 Case B

### B.2.1 Case B1

| ?- start.  
 Consulting the PLANNER...  
 Consulting the ONTOLOGY UPDATER...  
 Consulting ORS...  
 GOAL is: withChlorine(plantZero)  
 Translating ...  
 Need to find a plan ...  
 This is the plan: [sendWater(plantZero,x),sendChlorine(plantZero,x)]  
  
 Executing the plan ...  
 I'm going to ask plantTwo to perform sendWater(plantZero,x) for me  
 plantTwo asked me at(x,plantZero)  
 I told plantTwo no  
 The KIF ontology has been updated  
  
 sendWater(plantZero,x) failed. I am asking ORS for a diagnosis.  
 I am requesting a diagnosis ...I received a query about  
 at(x,plantZero), which I was not expecting to be asked about.  
 at(x,plantZero) has the same name as the precondition at(plantZero,x)  
 They also have the same arity (2)  
 Args class list is [contaminant,agent]  
 My unmatched classes: []  
 His unmatched classes: []  
 Right classes for args of this pred but in the wrong order.  
 Switch list is [[contaminant,agent],[agent,contaminant]]  
 I must check if the repair can be performed.  
 Protection used: (at,predicateAll,highProtection)  
 Sorry, the repair cannot be performed  
 ORS proposed the following diagnosis repair:

```

[at(plantZero,x),_885206,[switchArgs,at]]
I am asking SPA to perform switchArgs
plantTwo cannot do it
My protection is high.
I am asking SPA to try again.
This part of my ontology is protected.
plantTwo refused to implement the inverse.
I asked twice.
plantTwo will not be able to perform this type of repair for me.
GOAL is: withChlorine(plantZero)
Translating ...
Need to find a plan ...
This is the plan:
[sendWater(plantZero,x), sendChlorine(plantZero,x)]
Executing the plan ...
I'm going to ask plantSix to perform sendWater(plantZero,x) for me
plantSix asked me at(plantZero,x)
I told plantSix at(plantZero,x)
plantSix asked me criticalVolume(plantZero,_1383095)
I told plantSix criticalVolume(plantZero,200)
plantSix asked me usedVolume(plantZero,_1384198)
I told plantSix usedVolume(plantZero,100)
plantSix asked me class(plantZero,agent)
I told plantSix class(plantZero,agent)
plantSix asked me class(plantZero,agent)
I told plantSix class(plantZero,agent)
plantSix asked me class(plantZero,agent)
I told plantSix class(plantZero,agent)
plantSix asked me class(x,contaminant)
I told plantSix class(x,contaminant)
sendWater(plantZero,x) performed successfully
I'm going to ask plantFive to perform sendChlorine(plantZero,x) for
me
plantFive asked me at(plantZero,x)
I told plantFive at(plantZero,x)
plantFive asked me enoughWater(plantZero)
I told plantFive enoughWater(plantZero)
sendChlorine(plantZero,x) performed successfully
The KIF ontology has been updated
The plan is completed
The following actions have been performed:
[sendChlorine(plantZero,x),sendWater(plantZero,x),start]

```

To terminate the program, type "t."  
yes

### B.2.2 Case B2

— ?- start.

Consulting the PLANNER...

Consulting the ONTOLOGY UPDATER...

Consulting ORS...

GOAL is: withChlorine(plantZero)

Translating ...

Need to find a plan ...

This is the plan:

[sendWater(plantZero,x),sendChlorine(plantZero,x)]

Executing the plan ...

I'm going to ask plantTwo to perform sendWater(plantZero,x) for me

plantTwo asked me at(x,plantZero)

I told plantTwo no

The KIF ontology has been updated

sendWater(plantZero,x) failed. I am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about at(x,plantZero), which I was not expecting to be asked about.

at(x,plantZero) has the same name as the precondition at(plantZero,x)

They also have the same arity (2)

Args class list is [contaminant,agent]

My unmatched classes: []

His unmatched classes: []

Right classes for args of this pred but in the wrong order.

Switch list is [[contaminant,agent],[agent,contaminant]]

I must check if the repair can be performed.

Protection used: (at,predicateAll,highProtection)

Sorry, the repair cannot be performed

ORS proposed the following diagnosis repair:

[at(plantZero,x),\_827997,[switchArgs,at]]

I am asking SPA to perform switchArgs

plantTwo replied ok to repair request This part of my ontology is protected.

plantTwo agreed to implement the inverse.

GOAL is: withChlorine(plantZero)

Translating ...

Need to find a plan ...

This is the plan:

[sendWater(plantZero,x),sendChlorine(plantZero,x)]

```

Executing the plan ...
I'm going to ask plantTwo to perform sendWater(plantZero,x) for me
plantTwo asked me at(plantZero,x)
I told plantTwo at(plantZero,x)
plantTwo asked me criticalVolume(plantZero,_1097101)
I told plantTwo criticalVolume(plantZero,200)
plantTwo asked me usedVolume(plantZero,_1098140)
I told plantTwo usedVolume(plantZero,100)
plantTwo asked me class(plantZero,agent)
I told plantTwo class(plantZero,agent)
plantTwo asked me class(plantZero,agent)
I told plantTwo class(plantZero,agent)
plantTwo asked me class(plantZero,agent)
I told plantTwo class(plantZero,agent)
plantTwo asked me class(x,contaminant)
I told plantTwo class(x,contaminant)
sendWater(plantZero,x) performed successfully
I'm going to ask plantFive to perform sendChlorine(plantZero,x) for
me
plantFive asked me at(plantZero,x)
I told plantFive at(plantZero,x)
plantFive asked me enoughWater(plantZero)
I told plantFive enoughWater(plantZero)
sendChlorine(plantZero,x) performed successfully
The plan is completed
The following actions have been performed:
[sendChlorine(plantZero,x),sendWater(plantZero,x),start]
To terminate the program, type "t."
yes

```

### B.2.3 Case B3

```

| ?- start.
Consulting the PLANNER...
Consulting the ONTOLOGY UPDATER...
Consulting ORS...
GOAL is: withChloride(plantZero)
Translating ...
Need to find a plan ...
This is the plan:
[sendWater(plantZero,x),sendChloride(plantZero,x)]
Executing the plan ...

```

I'm going to ask plantTwo to perform sendWater(plantZero,x) for me  
 plantTwo asked me at(x,plantZero)  
 I told plantTwo no  
 The KIF ontology has been updated  
 sendWater(plantZero,x) failed. I am asking ORS for a diagnosis.  
 I am requesting a diagnosis ...I received a query about at(x,plantZero),  
 which I was not expecting to be asked about.  
 at(x,plantZero) has the same name as the precondition at(plantZero,x)  
 They also have the same arity (2)  
 Args class list is [contaminant,agent]  
 My unmatched classes: []  
 His unmatched classes: []  
 Right classes for args of this pred but in the wrong order.  
 Switch list is [[contaminant,agent],[agent,contaminant]]  
 I must check if the repair can be performed.  
 Protection used: (at,predicateAll,highProtection)  
 Sorry, the repair cannot be performed  
 ORS proposed the following diagnosis repair:  
 [at(plantZero,x),\_825983,[switchArgs,at]]  
 I am asking SPA to perform switchArgs  
 plantTwo cannot do it  
 My protection is high.  
 I am asking SPA to try again.  
 This part of my ontology is protected.  
 plantTwo agreed to perform the inverse.  
 I asked twice.  
 Translating ...  
 Need to find a plan ...  
 This is the plan:  
 [sendWater(plantZero,x),sendChloride(plantZero,x)]  
 Executing the plan ...  
 I'm going to ask plantTwo to perform  
 sendWater(plantZero,x) for me  
 plantTwo asked me at(plantZero,x)  
 I told plantTwo at(plantZero,x)  
 plantTwo asked me criticalVolume(plantZero,\_1092572)  
 I told plantTwo criticalVolume(plantZero,200)  
 plantTwo asked me usedVolume(plantZero,\_1093675)  
 I told plantTwo usedVolume(plantZero,100)  
 plantTwo asked me class(plantZero,agent)  
 I told plantTwo class(plantZero,agent)  
 plantTwo asked me class(plantZero,agent)  
 I told plantTwo class(plantZero,agent)

```

plantTwo asked me class(plantZero,agent)
I told plantTwo class(plantZero,agent)
plantTwo asked me class(x,contaminant)
I told plantTwo class(x,contaminant)
sendWater(plantZero,x) performed successfully
I'm going to ask plantFive to perform
sendChloride(plantZero,x) for me
plantFive asked me at(plantZero,x)
I told plantFive at(plantZero,x)
plantFive asked me enoughWater(plantZero)
I told plantFive enoughWater(plantZero)
sendChloride(plantZero,x) performed successfully
The KIF ontology has been updated
The plan is completed
The following actions have been performed:
[sendChloride(plantZero,x),sendWater(plantZero,x),start]
To terminate the program, type "t."
yes

```

## B.3 Case C

### B.3.1 Case C1

```

| ?- start.
Consulting the PLANNER...
Consulting the ONTOLOGY UPDATER...
Consulting ORS...
GOAL is: has(shoppingAgent,ourMutualFriend)
Translating ...
Need to find a plan ...
This is the plan:
[putInBasket(shoppingAgent,aiGroup,ourMutualFriend),
buy(shoppingAgent,ourMutualFriend)]
Executing the plan ...
I'm going to ask putInBasketAgentOne to perform
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) for me
putInBasketAgentOne asked me
class(aiGroup,shoppingGroup)
I told putInBasketAgentOne no
The KIF ontology has been updated
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) failed.

```

I am asking ORS for a diagnosis.  
I am requesting a diagnosis ...I received a query about  
class(aiGroup,shoppingGroup), which I was not expecting  
to be asked about.  
class(aiGroup,shoppingGroup) has the same name as the precondition  
class(aiGroup,group)  
They also have the same arity (2)  
My object is of the wrong class.  
putInBasketAgentOne expected an object of class shoppingGroup  
whereas I thought that an object of class group  
would be acceptable.  
group is a subclass of shoppingGroup so I need to be more specific about  
this object  
I must check if the repair can be performed.  
I am performing the repair  
ORS proposed the following diagnosis repair:  
[group,shoppingGroup,[precondAA,class]]  
This repair was performed.  
GOAL is: has(shoppingAgent,ourMutualFriend)  
Translating ...  
Need to find a plan ...  
This is the plan:  
[joinGroup(shoppingAgent,bookShopGroup),putInBasket(shoppingAgent,  
bookShopGroup,ourMutualFriend),  
buy(shoppingAgent,ourMutualFriend)]  
Executing the plan ...  
I'm going to ask joinGroupAgentOne to perform  
joinGroup(shoppingAgent,  
bookShopGroup) for me  
The KIF ontology has been updated  
joinGroup(shoppingAgent,bookShopGroup) failed.  
I am asking ORS for a diagnosis.  
I am requesting a diagnosis ...This plan failed immediately  
after a request was made to perform  
joinGroup(shoppingAgent,bookShopGroup)  
DIAGNOSIS: joinGroupAgentOne can't perform joinGroup(shoppingAgent,  
bookShopGroup)  
The appropriate repair has been performed  
ORS proposed the following diagnosis repair:  
[none,meta,[incorrectAgent,\_771047]] This repair was performed.  
GOAL is: has(shoppingAgent,ourMutualFriend)  
Translating ...  
Need to find a plan ...

This is the plan:

```
[joinGroup(shoppingAgent,bookShopGroup),putInBasket(shoppingAgent,
bookShopGroup,ourMutualFriend),
buy(shoppingAgent,ourMutualFriend)]
Executing the plan ...
```

```
I'm going to ask joinGroupAgentTwo to perform
joinGroup(shoppingAgent,bookShopGroup) for me
joinGroupAgentTwo asked me class(bookShopGroup,group)
I told joinGroupAgentTwo class(bookShopGroup,group)
joinGroup(shoppingAgent,bookShopGroup) performed successfully
I'm going to ask putInBasketAgentOne to perform
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) for me
putInBasketAgentOne asked me class(bookShopGroup,shoppingGroup)
I told putInBasketAgentOne class(bookShopGroup,shoppingGroup)
putInBasketAgentOne asked me registeredMember(shoppingAgent,bookShopGroup)
I told putInBasketAgentOne registeredMember(shoppingAgent,bookShopGroup)
putInBasketAgentOne asked me chooseItem(shoppingAgent,ourMutualFriend)
I told putInBasketAgentOne no
The KIF ontology has been updated
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) failed.
I am asking ORS for a diagnosis.
I am requesting a diagnosis ...I received a query about
chooseItem(shoppingAgent,ourMutualFriend), which I was not expecting
to be asked about.
No preconditions have the same name as chooseItem
DIAGNOSIS: Predicate anti-abstraction
chooseItem is a subtype of chooseThing
I must check if the repair can be performed.
Protection used: (chooseThing,predicateAll,highProtection)
Sorry, the repair cannot be performed
I knew the type
The appropriate repair has been performed
ORS proposed the following diagnosis repair:
[none,_1248978,[predicateAA,[chooseItem,chooseThing]]]
I am asking SPA to perform predicateA
putInBasketAgentOne wont bloody talk to me
putInBasketAgentOne cannot do it
My protection is high.
I am asking SPA to try again.
This part of my ontology is protected.
putInBasketAgentOne refused to implement the inverse.
putInBasketAgentOne will not be able to perform this type of repair for
```

me.

GOAL is: has(shoppingAgent,ourMutualFriend)

Translating ...

Need to find a plan ...

This is the plan:

```
[putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend),
buy(shoppingAgent,ourMutualFriend)]
```

Executing the plan ...

I'm going to ask putInBasketAgentTwo to perform

```
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) for me
```

```
putInBasketAgentTwo asked me class(bookShopGroup,shoppingGroup)
```

```
I told putInBasketAgentTwo class(bookShopGroup,shoppingGroup)
```

```
putInBasketAgentTwo asked me registeredMember(shoppingAgent,bookShopGroup)
```

```
I told putInBasketAgentTwo registeredMember(shoppingAgent,bookShopGroup)
```

```
putInBasketAgentTwo asked me chooseThing(shoppingAgent,ourMutualFriend)
```

```
I told putInBasketAgentTwo chooseThing(shoppingAgent,ourMutualFriend)
```

```
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)
```

```
performed successfully
```

```
I'm going to ask buyAgentOne to perform buy(shoppingAgent,ourMutualFriend)
for me
```

```
buyAgentOne asked me money(shoppingAgent,_1726678,_1726679)
```

```
I told buyAgentOne no
```

```
buyAgentOne asked me class(_1728831,currency)
```

```
I told buyAgentOne class(dollars,currency)
```

```
buyAgentOne asked me money(shoppingAgent,dollars,_1731177)
```

```
I told buyAgentOne no
```

The KIF ontology has been updated

```
buy(shoppingAgent,ourMutualFriend) failed.
```

I am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about

money(shoppingAgent,dollars,\_1861425), which I was not expecting to be asked about.

money(shoppingAgent,dollars,\_1861425) has the same name as the precondition money(shoppingAgent,100)

They have different arity (3 and 2)

```
[uninstantiated,currency,agent]money requires
```

```
an extra argument of type currency
```

I already know that currency is a subclass of thing I must check if the repair can be performed.

```
Protection used: (money,predicateArity,highProtection)
```

Sorry, the repair cannot be performed

ORS proposed the following diagnosis repair:

```
positional_anti_abstraction
```

I am asking SPA to perform propositionalA  
 buyAgentOne cannot do it  
 My protection is high.  
 I am asking SPA to try again.  
 This part of my ontology is protected.  
 buyAgentOne refused to implement the inverse.  
 I asked twice.  
 buyAgentOne will not be able to perform this type  
 of repair for me.  
 GOAL is: has(shoppingAgent,ourMutualFriend)  
 Translating ...  
 Need to find a plan ...  
 This is the plan:  
 [buy(shoppingAgent,ourMutualFriend)]  
 Executing the plan ...  
 I'm going to ask buyAgentTwo to perform  
 buy(shoppingAgent,ourMutualFriend) for me  
 buyAgentTwo asked me money(shoppingAgent,\_2314119)  
 I told buyAgentTwo money(shoppingAgent,100)  
 buyAgentTwo asked me inBasket(shoppingAgent,ourMutualFriend,\_2316955)  
 I told buyAgentTwo inBasket(shoppingAgent,ourMutualFriend,47899)  
 buyAgentTwo asked me class(shoppingAgent,agent)  
 I told buyAgentTwo class(shoppingAgent,agent)  
 buyAgentTwo asked me class(ourMutualFriend,item)  
 I told buyAgentTwo class(ourMutualFriend,item)  
 buyAgentTwo asked me class(shoppingAgent,agent)  
 I told buyAgentTwo class(shoppingAgent,agent)  
 buyAgentTwo asked me class(ourMutualFriend,item)  
 I told buyAgentTwo class(ourMutualFriend,item)  
 buy(shoppingAgent,ourMutualFriend)  
 performed successfully  
 The KIF ontology has been updated  
 The plan is completed  
 The following actions have been performed:  
 [buy(shoppingAgent,ourMutualFriend),  
 putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend),  
 joinGroup(shoppingAgent,bookShopGroup),start]  
 To terminate the program, type "t.  
 yes | ?-

### B.3.2 Case C2

— ?- start.

Consulting the PLANNER...

Consulting the ONTOLOGY UPDATER...

Consulting ORS...

GOAL is: has(shoppingAgent,ourMutualFriend)

Translating ...

Need to find a plan ...

This is the plan:

```
[putInBasket(shoppingAgent,aiGroup,ourMutualFriend),
buy(shoppingAgent,ourMutualFriend)]
```

Executing the plan ...

I'm going to ask putInBasketAgentOne to perform

```
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) for me
```

putInBasketAgentOne asked me class(aiGroup,shoppingGroup)

I told putInBasketAgentOne no

The KIF ontology has been updated

```
putInBasket(shoppingAgent,aiGroup,ourMutualFriend) failed.
```

I am asking ORS for a diagnosis.

I am requesting a diagnosis ...I received a query about class(aiGroup,shoppingGroup), which I was not expecting to be asked about.

class(aiGroup,shoppingGroup) has the same name as the precondition class(aiGroup,group)

They also have the same arity (2)

My object is of the wrong class.

putInBasketAgentOne expected an object of class

shoppingGroup whereas I thought that an object of

class group would be acceptable.

group is a subclass of shoppingGroup so I need to be more specific about this object

I must check if the repair can be performed.

I am performing the repair

ORS proposed the following diagnosis repair:

```
[group,shoppingGroup,[precondAA,class]]
```

This repair was performed.

GOAL is: has(shoppingAgent,ourMutualFriend)

Translating ...

Need to find a plan ...

This is the plan:

```
[joinGroup(shoppingAgent,bookShopGroup),putInBasket(shoppingAgent,
```

```

bookShopGroup,ourMutualFriend),buy(shoppingAgent,ourMutualFriend)]
Executing the plan ...
I'm going to ask joinGroupAgentOne to perform
joinGroup(shoppingAgent,bookShopGroup) for me
The KIF ontology has been updated
joinGroup(shoppingAgent,bookShopGroup) failed.
I am asking ORS for a diagnosis.
I am requesting a diagnosis ...This plan failed immediately
after a request was made to perform
joinGroup(shoppingAgent,bookShopGroup)
DIAGNOSIS: joinGroupAgentOne can't perform
joinGroup(shoppingAgent,bookShopGroup)
The appropriate repair has been performed
ORS proposed the following diagnosis repair:
[none,meta,[incorrectAgent,_771047]]
This repair was performed.
GOAL is: has(shoppingAgent,ourMutualFriend)
Translating ...
Need to find a plan ...
This is the plan:
[joinGroup(shoppingAgent,bookShopGroup),putInBasket(shoppingAgent,
bookShopGroup,ourMutualFriend),buy(shoppingAgent,ourMutualFriend)]
Executing the plan ...
I'm going to ask joinGroupAgentTwo to perform
joinGroup(shoppingAgent,bookShopGroup) for me
joinGroupAgentTwo asked me class(bookShopGroup,group)
I told joinGroupAgentTwo class(bookShopGroup,group)
joinGroup(shoppingAgent,bookShopGroup)
performed successfully
I'm going to ask putInBasketAgentOne to perform
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)
for me
putInBasketAgentOne asked me class(bookShopGroup,shoppingGroup)
I told putInBasketAgentOne class(bookShopGroup,shoppingGroup)
putInBasketAgentOne asked me registeredMember(shoppingAgent,bookShopGroup)
I told putInBasketAgentOne registeredMember(shoppingAgent,bookShopGroup)
putInBasketAgentOne asked me chooseItem(shoppingAgent,ourMutualFriend)
I told putInBasketAgentOne no
The KIF ontology has been updated
putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) failed.
I am asking ORS for a diagnosis.
I am requesting a diagnosis ...I received a query about
chooseItem(shoppingAgent,ourMutualFriend), which

```

I was not expecting to be asked about.  
 No preconditions have the same name as chooseItem  
 DIAGNOSIS: Predicate anti-abstraction  
 chooseItem is a subtype of chooseThing  
 I must check if the repair can be performed.  
 Protection used: (chooseThing,predicateAll,lowProtection)  
 Sorry, the repair cannot be performed  
 I knew the type  
 The appropriate repair has been performed  
 ORS proposed the following diagnosis repair:  
 [none, \_1248978, [predicateAA, [chooseItem, chooseThing]]]  
 I am asking SPA to perform predicateA  
 putInBasketAgentOne replied ok to repair request  
 This part of my ontology is protected.  
 putInBasketAgentOne agreed to implement the inverse.  
 GOAL is: has(shoppingAgent,ourMutualFriend)  
 Translating ...  
 Need to find a plan ...  
 This is the plan: [putInBasket(shoppingAgent,bookShopGroup,  
 ourMutualFriend),buy(shoppingAgent,ourMutualFriend)]  
 Executing the plan ...  
 I'm going to ask putInBasketAgentOne to perform  
 putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend) for me  
 putInBasketAgentOne asked me class(bookShopGroup,shoppingGroup)  
 I told putInBasketAgentOne class(bookShopGroup,shoppingGroup)  
 putInBasketAgentOne asked me registeredMember(shoppingAgent,bookShopGroup)  
 I told putInBasketAgentOne registeredMember(shoppingAgent,bookShopGroup)  
 putInBasketAgentOne asked me chooseThing(shoppingAgent,ourMutualFriend)  
 I told putInBasketAgentOne chooseThing(shoppingAgent,ourMutualFriend)  
 putInBasket(shoppingAgent,bookShopGroup,ourMutualFriend)  
 performed successfully  
 I'm going to ask buyAgentOne to perform  
 buy(shoppingAgent,ourMutualFriend) for me  
 buyAgentOne asked me money(shoppingAgent, \_2268628, \_2268629)  
 I told buyAgentOne no  
 buyAgentOne asked me class(\_2270797,currency)  
 I told buyAgentOne class(dollars,currency)  
 buyAgentOne asked me money(shoppingAgent,dollars, \_2273159)  
 I told buyAgentOne no  
 The KIF ontology has been updated  
 buy(shoppingAgent,ourMutualFriend) failed.  
 I am asking ORS for a diagnosis.  
 I am requesting a diagnosis ...

I received a query about money(shoppingAgent,dollars,\_2307576),  
 which I was not expecting to be asked about.  
 money(shoppingAgent,dollars,\_2307576) has the same name  
 as the precondition money(shoppingAgent,100)  
 They have different arity (3 and 2)  
 [uninstantiated,currency,agent]  
 money requires an extra argument of type currency  
 I already know that currency is a subclass of thing  
 I must check if the repair can be performed.  
 Protection used: (money,predicateArity,highProtection)  
 Sorry, the repair cannot be performed  
 ORS proposed the following diagnosis repair:  
 propositional\_anti\_abstraction  
 I am asking SPA to perform propositionalA  
 buyAgentOne replied ok to repair request  
 This part of my ontology is protected.  
 buyAgentOne agreed to implement the inverse.  
 GOAL is: has(shoppingAgent,ourMutualFriend)  
 Translating ...  
 Need to find a plan ...  
 This is the plan:  
 [buy(shoppingAgent,ourMutualFriend)]  
 Executing the plan ...  
 I'm going to ask buyAgentOne to perform  
 buy(shoppingAgent,ourMutualFriend) for me  
 buyAgentOne asked me money(shoppingAgent,\_2129888)  
 I told buyAgentOne money(shoppingAgent,100)  
 buyAgentOne asked me inBasket(shoppingAgent,ourMutualFriend,\_2132636)  
 I told buyAgentOne inBasket(shoppingAgent,ourMutualFriend,47899)  
 buyAgentOne asked me class(shoppingAgent,agent)  
 I told buyAgentOne class(shoppingAgent,agent)  
 buyAgentOne asked me class(ourMutualFriend,item)  
 I told buyAgentOne class(ourMutualFriend,item)  
 buyAgentOne asked me class(shoppingAgent,agent)  
 I told buyAgentOne class(shoppingAgent,agent)  
 buyAgentOne asked me class(ourMutualFriend,item)  
 I told buyAgentOne class(ourMutualFriend,item)  
 buy(shoppingAgent,ourMutualFriend) performed successfully  
 The KIF ontology has been updated  
 The plan is completed

The following actions have been performed:  
 [buy(shoppingAgent,ourMutualFriend),putInBasket(shoppingAgent,

```
bookShopGroup,ourMutualFriend),joinGroup(shoppingAgent,bookShopGroup),start]
```

To terminate the program, type "t."

```
yes  
| ?-
```



# Appendix C. Glossary

## **Abstraction**

**KIF** - Knowledge Interchange Format: first-order ontology language

## **Ontology**

## **Ontology mismatch**

**ORS** - Ontology Repair System

**PA** - see: Planning Agent

## **Planning Agent**

## **Refinement**

## **Service Providing Agent**

## **Signature**

**SPA** - see: Service Providing Agent

## **Theory**