Dynamic Ontology Refinement

Fiona McNeill

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2006

Abstract

Human communication is highly fault tolerant: although words and phrases that are not jointly understood are often used, a shared common language can be used to explain these differences. In agent communication, however, mutual comprehension usually depends on a perfect matching of messages to internal ontologies. Thus any kind of ontological mismatch will lead to communication failure, even though large sections of the ontologies may be common to both parties. Ontologies, once envisaged as definitive descriptions of what exists in a domain, are commonly not static but are continually updated and altered, both centrally and by individual users. As the environments in which agents interact become increasingly diverse and distributed, with agents being designed by a large number of different users, ontology mismatch becomes increasingly common.

Standard approaches to resolving this problem assume that the mismatched ontologies can be fully observed and often assume that it is desirable to match large sections, or even all, of the ontologies. However, this is not always a reasonable assumption, as many of these changes are not made public, and the computational cost of mapping entire ontologies is often prohibitive. We believe that it is more appropriate to assume that the ontologies of external agents are not available for observation, except for the specific parts of their ontologies revealed through normal agent communication. Consequently, a real-world solution, which we propose, is to patch specific instances of ontology mismatch when these particular mismatches lead to communication problems.

This thesis describes the development of ORS (Ontology Refinement System), a system designed to dynamically refine ontologies whenever mismatches lead to communication problems during agent interaction. ORS contains a framework for agents to diagnose and refine ontological mismatch, integrated within an environment where

planning agents can use this ability to achieve goals that would otherwise have been unreachable. These abilities are evaluated against genuine examples of ontological mismatch to demonstrate that they are useful and can be successfully performed.

Acknowledgements

I am extremely grateful and glad to have a supervisor who is as knowledgeable, imaginative, dedicated, astute and personable as Alan Bundy. I have often been told that he's the best supervisor one could have and I have always fervently believed it. I am very thankful also to my second supervisors, Chris Walton (2003-2005) and Marco Schorlemmer (2001-2003), for their encouragement, enthusiasm, tireless rereading of drafts, and their thoughtful and helpful comments.

I would like to thank my examiners, Prof. Frank van Harmelen and Dr Dave Robertson, whose thoughtful criticism created a challenging yet encouraging viva and whose comments have improved the quality of the thesis.

I have enormously enjoyed being a member of the DReaM group, and an occasional interloper of the SSP group. I am grateful to them for help and encouragement, and distraction and drinks as necessary, and especially to Lucas and Graham, who understand the things I don't. Also to the Others - Alison, Dan, Joe, Paul and Seb - for providing perspective and gin.

I would like to thank my family for always being supportive. And Hamish, for everything.

Publications

Some of the work in this thesis has previously been published, in [McNeill et al., 2003a, McNeill et al., 2004a, McNeill et al., 2004b, McNeill et al., 2004c, McNeill et al., 2005, McNeill et al., 2003b].

These papers are available on the project website:
`http://dream.inf.ed.ac.uk/projects/dor/`.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Fiona McNeill*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A central part of the ability to communicate is a shared understanding of the world. An important aspect of this shared understanding is the use of a common language in which to represent knowledge about the world. It is a widely held assumption that humans who share a common language can interact successfully with one another. However, two humans who claim to share a language may not always have identical ways of describing the world. It is a common experience in human interaction that people from a different background, or with different levels of expertise in a subject area, may have problems with communication. For example, a specialist describing his area of expertise may use many words and phrases that are not part of the vocabulary of a non-specialist; if he wishes to describe his ideas to a general audience, he must explain what these words and phrases mean in terms of commonly-used words and phrases that he can expect the whole audience to understand. Equally, a speaker of American English and a speaker of British English may believe they speak the same language, but often they will need to translate words and idioms into a shared part of the language before they can be fully understood. In general, the wider the gap between the two dialects, the smaller the shared understanding, and thus the greater the difficulty in communication.

Likewise, in an agent communication system, a widely held assumption is that the agents will have identical ways of describing the world and thus can communicate successfully. For an agent, its language, or way of describing the world, is determined by its ontology. A common restriction of agent systems is that agents must have identical ontologies: the way in which they represent the world must be the same. However, this is not always a reasonable assumption. Large scale systems that facilitate multi-agent communication, such as the Semantic Web, are far more successful if they place less rigid constraints on the agents that make use of them. It is essential to such systems that agents are created and owned by many different users, and forcing tight ontological constraints on potential users greatly reduces the usability of the system.

We believe that these tight ontological restrictions are not necessary. As with human communication, agents that share a large part of their ontology but differ in some aspects can use this shared part of their ontology as a basis for discussing those parts in which they differ. Agents can thus refine their ontologies so that a representational difference that led initially to failed or problematic communication can be resolved so that communication proceeds smoothly.

For example, an ontology that had been developed in the US might have a *Money* predicate that made no reference to currency, because it was implicitly assumed that the currency would always be dollars. However, if this ontology is made use of in Europe, it might be considered necessary to include some currency information related to the *Money* predicate, since the implicit assumption no longer holds. Thus these two versions of the ontology would have conflicting representations for the predicate *Money*; perhaps: *(Money ?Amount ?Owner)* and *(Money ?Amount ?Owner ?Currency)*.
This difference will cause problems when agents using these two ontologies are attempting to communicate. However, there is sufficient information in both ontologies to determine that these two uses of money are related, and to determine the link between them. Thus, if agents are equipped with the ability to identify and solve these

problems, it is possible for them to represent money in these different ways and still communicate successfully.

This project investigates the techniques we might need to build into an agent communication system to facilitate this kind of diagnostic and refinement process, and builds a system, ORS (Ontology Refinement System), that can automatically perform this task.

1.1 Motivation

Ontological mismatches between agents can be a serious problem in agent communication, particularly where agents come from different sources, different users or are adapted to different tasks. Even where agents' ontologies have originated from the same source, updates and alterations may make communication impossible. At the same time, since it is advantageous that agents are adapted to their particular tasks, forcing all agents to use identical ontologies is not a sensible or practical solution, nor are there any established mechanisms to enable this. In addition, large scale systems like the Semantic Web will only become successful through attracting many and diverse users. This can only be practically achieved through making it easy for people to use the system and by placing as few restraints on their joining as possible; this is the approach that is behind the enormous success of the World Wide Web.

A key concept of the Semantic Web is the automated processing of ontological knowledge by agents and services. [Berners-Lee et al., 2001] outlines how these are central to the role of the Semantic Web, and how they require ontological matching to perform appropriately. However, the nature of the Semantic Web means that strict controls on these ontologies are impractical: it must allow for, amongst other things, partial information, erroneous information and the evolution of ontologies [Koivunen and Miller, 2001]. Additionally, it is not always reasonable to assume that complete ontological information is available for agents encountered on the Semantic Web. Their ontologies

may be security or commercially sensitive, or they may simply not have been made publicly available. All of these requirements cause difficulties for agent communication. In this project, we develop a method of dealing with these ontological mismatches without enforcing tight ontological controls on agents and without assuming that any information about other agents' ontologies, other than what can be gleaned during agent communication, is available. It is intended to automatically solve a subset of the many challenges surrounding ontology mapping, merging and alignment, which are all essential for communication in multi-agent systems. It allows agents to learn about the area in which they are currently active, and thus be more capable of acting successfully within it, without receiving any extraneous information. This is particularly useful in complex and dynamic domains, where a complete and up-to-date representation of the entire domain may be intractable or undesirable. We have developed a system, ORS, with which agents can identify the specific ontological causes of communication failure and adapt these parts of their ontologies dynamically during interactions so that communication becomes possible.

1.2 Context of the System

"To learn, a learner needs to formulate plans, monitor the plan execution to detect violated expectations, and then diagnose and rectify errors which the disconfirming data reveal." Frederick Hayes-Roth [Hayes-Roth, 1983]

This project is concerned with the resolution of the problem of ontological mismatch within a planning context. An agent forms plans to achieve a goal based on its understanding of the domain, and then attempts to execute these plans through communication with other agents. Planning in complex and dynamic environments is very difficult because any incomplete, incorrect or out-of-date information can cause an inexecutable plan to be developed. However, by adopting our approach to ontological refinement,

these cases of plan execution failure can be considered to be opportunities to learn more about the domain. Information about the cause of failure is extracted from observation of how and why the plan failed, augmented by further communication with the other agents involved. Once the point of failure has been located, refinement techniques are implemented to fix the problem, and a new plan is developed using the updated ontology. This plan is more likely to succeed than the previous plan, although success is not guaranteed. This procedure is repeated until the goal is successfully reached, or until it becomes impossible to form a plan to achieve the goal from the updated ontology.

We consider that there are three essential elements to creating such a dynamic ontology refinement system:

1. the ability to link the relevant information about the underlying ontology to the plan;
2. the ability use this information to locate the exact source of the problem;
3. the ability to select and apply appropriate techniques for altering the ontology.

These abilities are implemented in different sub-systems within the dynamic ontology refinement system [McNeill et al., 2003a]: the *planning system*, which forms and interprets plans; the *diagnostic system*, which locates the source of the problem; and the *refinement system*, which implements the necessary changes to the ontology. The environment in which plans are executed, and the necessary communication carried out, is part of the *agent communication system.*

Such a system is inherently limited: it is very difficult or perhaps impossible to define every possible kind ontological mismatch, and additionally it may be impossible to diagnose differences that are chaotic in nature; for example, names that are altered arbitrarily. Our system relies on there being some method to the changes made: for example, an argument's class restricted to a subclass. Additionally, we have made many

assumptions in order to simplify the problem to the extent where we can produce a working system. Some of these assumptions are necessary because they concern parts of ORS that are not the focus of the research, and thus should not detract from the main work of the thesis. These assumptions are mostly concerned with agent behaviour and interaction. For example, we have assumed that agents are honest and helpful. We have assumed that they will: perform tasks they are capable of performing, if relevant preconditions are fulfilled; give truth values for specific statements; instantiate variables in a specific predicate, according to their ontology; and give class information about specific objects. We do not assume that they will reveal large sections of their ontology; since these agents are from different sources, we feel that this is too high a level of trust to assume. Additionally, this approach must work with all agents that are encountered, not merely those that are specifically programmed to behave appropriately, and therefore we cannot assume that agents are able to reveal their ontologies in this way. These refinement techniques are intended for agents which are using similar ontologies that differ slightly; for example, one agent has an ontology that has been updated or altered, or perhaps one agent is using an old version of an ontology. They are not intended for facilitating communication between agents with completely different ontologies; there must be a reasonable degree of shared ontology from which the differences can be inferred. We have also assumed that the agents that are attempting to execute the plans are willing to believe the other agents with which they are communicating and alter their ontology accordingly. There is currently no analysis of which of the agents is more likely to have a correct ontology. The assumptions are explained in more detail in Section 6.2

These assumptions have been made in order to avoid too much of the research being focused on non-central issues. In addition, other assumptions are necessary to limit the scope of the problem and make it more tractable. These are concerned with the domain in which we are investigating the problem and the ontological representations we expect to encounter; they are detailed in Chapter 4.

The system could later be made more sophisticated to eradicate these assumptions, but in the meantime they reduce the problem to a solvable subset. Discussions about how to extend this subset can be found in Chapter 8.

1.3 Ontologies

There is some ambiguity and dissension over the meaning of ontology. We define here the way in which we use the term. Further discussion of other uses of the term, and the field of ontologies in general, can be found in Section 2.2.

We use the term *ontology* to refer to the whole of the domain knowledge. This consists of two parts:

- the *signature*, which describes the representation language in which the ontology is written: what predicates exist, what arity they have, what classes their arguments have, the class hierarchy and so on;
- the *theory*, which contains the formulae written in the representation language.

We are primarily interested in mismatches in the signature. However, alterations in the signature will normally also entail alterations in the theory, where the particular signature object is instantiated. There are many different levels of expressiveness which can be contained in an ontological representation. Standard ontology representations range from full first-order to considerably restricted representations; higher-order ontological representations are not commonly used as reasoning with them is intractable. This project focuses exclusively on first-order ontologies.

ORS works with a restricted version of KIF (Knowledge Interchange Format). The motivation for this choice is detailed in Section 4.5, and further detail about KIF can be

found in Section 2.2.1. KIF ontologies include classes, relations, functions, individuals and axioms. Functions are relations for which, if there are n arguments, and $(n-1)$ arguments are instantiated, then the nth argument is determined. Axioms can be thought of as implication rules that have a conjunction of relations determining when the rule is applicable, and a conjunction of relations describing the situation after the rule has been applied. Much of the research of the project is focused on identifying what potential mismatches could arise between these ontological objects.

1.4 Results

In this thesis, we outline potential ontological mismatches that we believe can occur between ontologies represented in first-order logic. We have implemented the ability to detect most of these in ORS; for those that are not implemented, we provide explanations for why this has not been done. ORS is demonstrated to successfully perform all of these potential mismatches except for those which we have justified reasons for not implementing. We have evaluated ORS against off-the-shelf ontologies for which we have access to different versions, and have demonstrated that ORS can successfully refine the mismatches that occur between these different versions in a high percentage of cases. For those cases that it cannot successfully perform, we have provided an explanation for why this has not been achieved. This evaluation process is outlined in Chapter 7.

1.5 Research Aims

The aim of the project is to demonstrate the following:

> Using dynamic ontology refinement to locate and correct ontological mismatches between agents can enable successful communication which would otherwise be impossible.

The area of ontological mismatch is a key issue in agent technology, particularly in application areas such as the Semantic Web. In a more general context, the issues surrounding learning through experience, and using failure of expectations to develop a more sophisticated view of the world have been central to Artificial Intelligence for much of its history. Even when these issues are limited to the specific context of agent interaction, they still generate huge and difficult problems on which a large number of people in the agent community are working. This thesis does not attempt to solve such issues in a complete and general manner; rather, we have developed a new approach to them. A theory has been developed about what kinds of mismatches one might expect to encounter, how these could be recognised and what kind of techniques would help to fix them. We have created a system to illustrate the implementation of this theory within a restricted context.

This system can, for agents using a certain platform and communication language, identify ontological mismatch and, in many cases, diagnose the cause of the failure and refine it appropriately. This is not guaranteed to work in all situations, since in some cases there is not enough information for fully automated ontology refinement. However, we consider the main contribution of this thesis to be the new approach to these issues that we present here. Although the actual implementation is limited, the theory behind it could potentially apply to much more sophisticated systems.

The key aims, then, of this project are:

1. To provide a framework in which agents with first-order, largely similar ontologies can diagnose ontological mismatches between them;

2. To integrate this framework into a system, ORS, that enables an environment where planning agents can use this ability to reach goals that would otherwise have been unreachable. This system must be fully automated;

3. To evaluate these abilities against genuine examples of ontological mismatches, to demonstrate that these abilities are useful and can be successfully performed.

All of these aims have been achieved; this is discussed further in Chapter 9.

1.6 Organisation of Thesis

The organisation of the thesis is as follows:

Chapters 2 and 3 review the relevant literature. Chapter 2 outlines the background of this project, describing the AI techniques on which this project is building, whilst Chapter 3 describes the work that has been done to achieve similar aims to those of this project, in order to explain how our work fits in with the field.

Chapter 4 provides an overview of ORS. It describes the various sub-systems and explain how they interact.

Chapter 5 explains in more detail the diagnostic and refinement aspects of ORS. This constitutes the heart of the research.

Chapter 6 details the other sub-systems of ORS; it explains their importance to ORS and describes the theory on which they are based and the implementational issues surrounding them.

Chapter 7 provides an evaluation of ORS, both from a practical viewpoint (how well does it work?) and a theoretical viewpoint (how useful is it?). This chapter also provides a discussion of work that is closely related to the work of this project to assess the contribution of ORS.

Chapter 8 discusses directions in which this research could be expanded.

Chapter 9 summarises the thesis and draws conclusions.

Chapter 2

Background

The scope of this project is broad, and there are many different areas in Artificial Intelligence that are of some relevance to the work; in particular: agents, ontologies, the Semantic Web, peer-to-peer systems and planning. In this chapter, we introduce the fields on which our project depends, in order to ensure that the reader has a sufficient understanding of the technology we are making use of. The fields described in this chapter are not those in which the focus of the project is principally concerned, but rather the background that provides a platform from which ORS can operate and the motivation for the functionality of ORS. The fields to which this project makes a direct research contribution are discussed in Chapter 3.

After each section in this chapter, we explain how the field is of relevance to the project, firstly by explaining what aspects of the field we need to use in the system, and secondly by explaining how our research contributes to that field. In many cases, these contributions are not significant; the major contributions of the project lie in the fields discussed in Chapter 3.

2.1 Agents

There is much debate over the issue of exactly what an agent is and there is no precise, universally accepted definition [Franklin and Graesser, 1997]. However, in Artificial Intelligence an agent is generally considered to be a computer system that is situated in some environment and that is capable of autonomous action in this environment in order to meet its design objectives [Wooldridge, 2002]. Intelligent agents are generally considered to exhibit [Wooldridge, 1999, Wooldridge and Jennings, 1995]:

- **Autonomy:** agents act without direct intervention and have some control over their actions and internal states;
- **Reactivity:** agents respond to their environment in suitable ways;
- **Pro-activeness:** agents exhibit goal-directed behaviour by taking the initiative;
- **Social ability:** agents interact with other agents and possibly humans to satisfy design objectives.

In addition, agents might be assumed to exhibit the following:

- **mobility** - the ability to move around an electronic network;
- **veracity** - the propensity to avoid communicating information that is known to be false;
- **benevolence** - the propensity for an agent to try to do what is asked of it;
- **rationality** - the propensity for an agent to act in order to achieve its goals to the best of its ability and understanding;
- **learning** - the ability to infer new or improved information about the domain as a result of interactions within it.

These attributes are not essential to agents; in particular, benevolence may not occur if agents have conflicting goals, which is certainly possible if the agents are from different sources.

Complex systems consist of a number of related sub-systems which are modular in terms of the objectives they achieve and have multiple loci of control [Jennings, 1999]. Thus it is useful to consider each part of a system to be an agent with its own objective(s) and autonomy. Since it is impossible to know *a priori* about all potential links, it is helpful to endow the components with the ability to make decisions about their behaviour.

However, there are many difficult issues surrounding the use of agents [Weiß, 1999], which include:

- agent communication (see Section 2.1.1);
- how agents should represent and reason about the knowledge, plans and actions of other agents;
- how agents should represent and reason about the interaction process;
- how agents should recognise and reconcile disparate viewpoints and conflicts and how to synthesise views and results;
- how agents can negotiate and make contracts with other agents;
- how to formally describe multiagent systems and interactions among agents;
- how to realise 'intelligent processes' such as problem solving, planning, decision making and learning in multiagent contexts.

2.1.1 Agent Communication

One of the basic functions of an agent is the ability to communicate, primarily with other agents. Enabling successful communication between diverse agents can be difficult, and a large amount of research within the agent community is focused on this issue. In keeping with the idea of anthropomorphising computer systems to produce agents, agent communication theory is based on spoken human communication theory [Huhns and Stephens, 1999], which is formally described in speech act theory [Austin, 1962, Searle, 1970].

There are three levels of structure involved in agent communication:

1. **Message content**, which contains the locution which the agent wishes to make; for example, "it is cold";

2. **Performatives**, which describe what kind of message is being passed; for example:

   {**Tell:** "it is cold"}, or

   {**Query:** "it is cold"}.

   Common examples of performatives include *confirm, disconfirm, promise, tell, request, report*, etc. Performatives make explicit what the purpose of the message is, allowing agent communication to avoid the confusion and ambiguity that is often found in human speech. In agent communication, messages must always have some purpose. Whilst the message content can be viewed as a locution, the performative can be viewed as an illocution: it conveys the intended meaning of the message. Performatives are analogous to real-world actions.

3. **The Protocol**, which determines how the conversation is to be carried out between the agents. This dictates the exact structure of each message, and determines which particular speech acts are permissible to agents in a given situation.

The permissible performatives and their precise meanings are defined within the protocol, as are the parameters that are permissible or necessary within the message.

The most common agent communication protocols are FIPA-ACL [FIPA-ACL, 2002] and KQML [KQML, 2005].

KQML

KQML (Knowledge Query and Manipulation Language) is a protocol for exchanging knowledge and information. There are several different versions of KQML, each of which contain different performatives. This proved to be a disadvantage of KQML, because it lead to situations where agents with different versions of KQML were not able to interact successfully, because they did not have the same performatives [Wooldridge, 2002].

KQML depends on the notion of a *virtual knowledge base* (VKB). In order for agents to communicate successfully using KQML, it is not necessary for them to have the same internal representation of knowledge. However, communication is much facilitated if the agents behave as if they had some internal representation of knowledge. The VBK is the attribution of knowledge by the agents to other agents.

KQML has been very successful, and its introduction was an extremely significant development in multiagent system development. However, it has received criticism. The semantics are not rigorously defined and sometimes leave meaning open to interpretation; there are performatives that are not represented at all in KQML, and the performative set is considered to be too large and rather *ad hoc*.

FIPA-ACL

FIPA (Foundations for Intelligent Physical Agents) is an influential organisation aimed at producing standards for the interoperation of heterogeneous software agents [FIPA, 2005]. Their Agent Communication Language (ACL) has become a standard agent protocol. FIPA describes FIPA-ACL as *"A language with precisely defined syntax, semantics and pragmatics that is the basis of communication between independently designed and developed software agents"* [FIPA, 1997].

It is superficially similar to KQML; the relationship between these two protocols is discussed in [FIPA, 1997]. The main difference between them is in the performatives. FIPA-ACL addressed many of the criticisms of KQML, such as the lack of formal semantics. It contains twenty-two performatives, which can be grouped into five categories: *passing information, requesting information, negotiation, performing actions* and *error handling*.

Agent Ontologies

In order to communicate successfully, it is not only necessary for agents to have protocols and performatives that they can both understand, but also for the message content to use terminology which they both understand. The knowledge of an agent about the world, and the terminology which is used to represent this, is contained in the agent's ontology. This is discussed in Section 2.2.

2.1.2 Multiagent Systems

In order to be able to function in a useful way, agents normally exist within multiagent systems [Huhns and Stephens, 1999]. It is within these systems that they perform their tasks and attempt to achieve their goals by cooperating with other agents and gaining

access to services that are controlled and maintained by other agents. Agents represent their users or owners, and receive their goals from them. To a large extent, they also receive their ontologies and protocols from their owners or designers, although these may be altered to some extent by the agents, independent of their owners, if they are learning about the domain. The concept of agents and multiagent systems provides a basis for large-scale automated interaction between users who are geographically, functionally and intentionally diverse. In order to enable easy visualisation of the interactions that are possible within a multiagent system, they are often modelled on standard platforms for human interactions: for example, an auction house or a market place. This anthropomorphisation of agents provides a simple way to understand and talk about what is happening within these large, complex systems, and hence facilitates their design. It allows terms such as cooperation, negotiation, intention, and so on, to become meaningful in the context of computer systems interacting, which allows users to model their own desires and intentions easily.

In order for multiagent systems to function, we require [Durfee, 1999]:

- **Coherence** - agents need to want to work together;
- **Competence** - agents need to know how to work well together.

Coherence does not rule out the possibility of agents being competitive or self-interested; in fact, these aspects are central to many agents in multiagent systems, such as those involved in commercial activities. If the agents do not wish to work together purely for the benefit of others then there must be other reasons for them to wish to interact: for example, their aims are served through interaction with particular agents, or they gain more benefit from collective efforts, or there are disincentives introduced to the system for agent individualism.

Competence is a fairly simple issue if the agents are all designed to the same pattern. However, if agents use different languages, protocols or ontologies then this issue can

become extremely complex. The aim of this project could be considered as an attempt to ease the problem of competence in a multiagent system where the agents do not all share the same ontology.

2.1.3 Relevance to the Project

- **What the project requires from agent technology:**

  The processes of diagnosis and refinement, which are the central interest of this project, are set, for this project, within an agent communication system. The refinements are being performed on agent ontologies, and are necessary due to plan execution failure within the multiagent system. Thus the project depends heavily on the ability to build agents and a multiagent system within which they can interact. However, since the multiagent system is the platform for the more central focus of the project, rather than the central focus itself, we have kept it as simple as possible, and have not investigated the more complex problems that multiagent systems entail.

- **What the project contributes to agent technology:**

  The central aim of ORS—to enable communication between agents that would otherwise be unable to communicate due to ontological mismatch—is a useful addition to communication within a multiagent system. Since we have not attempted to do anything complex with agents in this project, but we have rather borrowed the research that allows us to build a simple agent communication system, we have not otherwise contributed anything to this field.

2.2 Ontologies

The concept of ontologies is central to our work. In this section, we introduce the notion of ontologies, to ensure that the reader has a sufficient understanding of the underlying concepts on which this project is based. We do not, in this section, examine the sub-issues of ontology mapping and refining, which are closely related to the work of this project; these are explored in Chapter 3.

The AI term *ontology* is derived from the philosophical term of the same name, meaning a systematic account of existence. Ontology, in philosophy, is concerned with what exists and how things that exist are related to one another, and is clearly distinct from epistemology, meaning what is known about or believed to exist. The word ontology was adapted by AI to refer to a formal specification of what exists: for example, formalising concepts, objects and relationships between them. Because ontologies must completely and explicitly describe what exists, they are not universal, describing everything that exists, but instead describe a specific domain. Such domains are typically fairly small, as it is laborious and difficult to describe domains completely and explicitly, and extremely large ontologies are difficult to reason with.

An ontology could be considered to be a description of a domain: what exists in a domain is defined by what it is in the ontology. However, as agent technology has developed, and multiagent systems commonly consist of agents owned and designed by multiple users, it is often the case that the ontologies of agents diverge; both the knowledge contained in an ontology and the representation of that knowledge may differ due to the different aims and needs of the agents. In such situations, an agent's ontology cannot be considered to be an objective definition of what exists, but rather as reflecting the agent's understanding or interpretation of what exists. In many situations, ontologies are considered to be shared knowledge; that is, it is the manifestation of a shared understanding of a domain that is agreed between a number of agents;

agents that diverge from this ontology can be said to be mistaken and in some sense objectively wrong. However, in peer-to-peer systems, there is no definitive version of reality; rather, if the differences cause problems, the agents must negotiate as to which, if any, of their divergent ontologies to accept, or how these ontologies can be merged. In this context, the meaning of ontology is thus closer to the philosophical term epistemology than it is to the philosophical term ontology.

There are, in fact, many different definitions for ontology that are in use today. This comes about partly due to the fact that people with diverse backgrounds and interests are engaged in developing and using ontologies, and naturally create ontologies that suit their own particular needs. Gruber defined an ontology as *"an explicit specification of a conceptualisation"* [Gruber, 1993], which was expanded by Borst *et al* [Borst et al., 1997] to prescribe that the specification should be formal and the conceptualisation should be shared (note that this does not fit with how ontologies are regarded in peer-to-peer systems: although, in such systems, there may be a large amount of shared knowledge present, it is not assumed that the entire ontology must be shared or agreed on). An ontology can be considered to be *"a specification of the objects, concepts and relationships in an area of interest"* [Huhns and Stephens, 1999], and thus provides potential terms for describing knowledge about the domain [Chandrasekaran et al., 1999]. Kalfoglou examined many of these similar but slightly different definitions [Kalfoglou, 2002] and produced a summary of an ontology as *an explicit representation of a shared understanding of the important concepts in some domain of interest.*". He considered it to be central to the role of an ontology that it supports knowledge sharing within and among groups of agents.

At its most general level, an ontology can be considered to be the vocabulary, or signature, that is used to describe the world. Whether, for example, money is considered to be a concept that requires only one argument: *(Money ?Amount)*; or two arguments: *(Money ?Amount ?Owner)*; or three arguments: *(Money ?Amount ?Owner ?Currency)*,

is determined by the ontology. An ontology also contains a class hierarchy, which is a taxonomy describing how classes of objects are related to one another. The signature is used to create a theory, which consists of facts about the world that are written using the vocabulary: for example, *(Money 10 Agent1 Sterling)*. The term ontology is used either to refer to both the signature and the theory, or just to the signature, with the theory being considered to be something outwith the ontology. In this project, we use the former definition, as described in Section 1.3.

Kalfoglou and Schorlemmer [Kalfoglou and Schorlemmer, 2003b] define an ontology as $O = (S, A)$, where S is a signature and A is a set of axioms (or theory). This corresponds with the definition of ontology we use.

When an agent *commits* to an ontology, this means that its observable actions are consistent with the definitions in the ontology [Gruber, 1995]. An ontological commitment is an agreement to use a vocabulary in a way that is consistent with respect to the theory that specifies the ontology.

An important role of ontologies is facilitating the reuse and sharing of knowledge [Gruber, 1991]. Building large knowledge bases is a difficult and costly process; it is therefore essential that the effort put into this can be used more than once in more than one particular situation. Traditional knowledge bases lacked the ability to facilitate this: reusing can only be done by adopting the entire representation and programming environment of the existing KB. Since ontologies build up a vocabulary of terms to describe a domain, they are thus more amenable to reuse and sharing. Many theories can be developed for a single signature, so that a domain description can cover many instantiations of that domain. Additionally, ontologies facilitate the adaptation of domain knowledge by users who wish to alter or add new concepts or relationships. Online systems have been created for aiding the reuse and sharing of ontologies, a well-known example of which is Ontolingua [Farquhar et al., 1996, Gruber, 1992, Gruber, 1993]. Ontolingua is a system for analysing and translating ontologies. It aims

to describe ontologies in a form that is compatible with multiple representation languages, and has a translation mechanism so that Ontolingua ontologies can be shared by users requiring CLIPS, CLM, EpiKit, Interface Description Language, LOOM or Prolog Syntax. The ontologies used in this project were written using Ontolingua.

2.2.1 Ontological Representation

One of the most crucial decisions that must be made in ontology construction is what representation will be used. There are two conflicting issues at stake: how to allow expressive representation, and how to enable tractable reasoning. The first issue is best served by using an extremely expressive representation, such as full first-order logic, and this is the basis of many popular ontological representations, such as KIF. However, full first-order logic is not decidable and consequently can be difficult to reason with. Much work has therefore been invested into finding a better compromise between the two issues: to find a subset of first-order logic that retains a high level of expressiveness, but at the same time is not undecidable or difficult to reason with.

Description Logics

Description Logics (DLs) represent some of the more successful attempts to balance the need for tractability against the need for expressivity. DLs are subsets of first-order logic; for example, the DL ALC corresponds to the fragments of first-order logic obtained by restricting the syntax to formulae containing two variables [Nardi and Brachman, 2003]. ALC is the smallest propositional closed DL, and can in some ways be considered to be the basic DL. It is lacking in expressive power; for example, it does not have appropriate operators to deal with inequality and equivalence. Many other DLs are extensions to ALC, and these have greater expressivity. However, they suffer from increased computational complexity, making them less tractable.

DLs grew out of earlier work on representing knowledge through semantic networks [Quillian, 1967] and frames [Minsky, 1985]. The essential components of a DL knowledge base are individuals, concepts and roles that relate them. Concepts can be equated to classes, describing the common properties of a collection of individuals, and are thus unary relationships. Roles are binary relations between objects. New concepts and roles can be defined using language constructs that are defined by each DL, such as intersection, union and role quantification. The main reasoning tasks are classification, satisfiability, subsumption and instance checking. A DL ontology is split into two sections: the T-box and the A-box. The basic form of declaration in a T-Box is a concept definition, where new concepts are defined in terms of previously defined concepts. They contain the terminological and background knowledge and can be considered to be equivalent to our definition of signature. The A-Box contains assertions about individuals, called membership assertions, and can be considered to be equivalent to our definition of theory.

DLs are widely used. They are the basis for most of the web ontology representations discussed in Section 2.3, and have been used in a variety of applications, including conceptual modelling, information integration, query mechanisms, view maintenance, software management systems, configurations systems and natural language understanding [DL, 2004].

KIF

Knowledge Interchange Format (KIF) is perhaps the most popular full first-order ontological representation format. KIF was designed for use in the interchange of knowledge among disparate computer systems, which have been created by different programmers at different times and in different languages [Genesereth and Fikes, 1992]. There are three essential features to KIF:

- It has declarative semantics;

- It is logically comprehensive and provides for the expression of arbitrary first-order logical sentences;
- It provides for the representation of knowledge about knowledge.

Ontologies written on the Ontolingua server are written in a specialised representation language which is based on KIF, together with the Ontolingua frame-ontology.

Semantic Web Representations

Ontologies are central to the Semantic Web. The growth of the Semantic Web has lead to an increase in the importance of developing ontology research, and the divergent, distributed and large-scale nature of the Semantic Web has meant that the sophistication and functionality demanded from ontological representations has greatly increased. The resulting representations are discussed in Section 2.3.1.

2.2.2 Ontologies and Contexts

A different approach to the problem of describing information semantics is through the use of *contexts*. The difference between ontologies and context can be summarised as follows [Bouquet et al., 2004]:

- **Ontologies** are intended to be *shared* and to represent a view that is common to different parties;
- **Contexts** are not intended to be shared, but rather are *local*, and represent a personal view that is particular to an individual.

A context is a subset of a knowledge base that is relevant in a particular instance [Giunchiglia, 1993]. It is a theory of the world that encodes an individual's subjective perspective to the world.

It is clear that ontologies are more useful than contexts in many situations because they enable the sharing of knowledge, which facilitates communication and information exchange. However, ontologies depend on there being a consensus of opinion about the knowledge being represented, and on each agent that accesses that ontology requiring only knowledge that is shared, and not keeping any personal information to itself. Additionally, they can be very difficult to create and maintain. The use of contexts can alleviate some of these problems; agents can have private information, and contexts are easy to build and maintain because it is not necessary to obtain a consensus on how information should be represented. However, inter-context communication requires there to be mappings from the context of one agent to the context of another.

[Bouquet et al., 2004] proposes the integration of contexts and ontologies to produce *contextual ontologies*, which are considered to be ontologies that are kept local, and thus not shared with other ontologies, but whose content is put in relation with the content of other ontologies via explicit mappings. Bouquet *et al* have developed Context OWL (C-OWL) to allow for the representation of contextual ontologies. C-OWL is an extension of OWL (see Section 2.3.1), with the addition of *bridge rules*, which allow concepts, roles and individuals in one ontology to be related to those in another. Thus a contextual ontology is an OWL ontology embedded in a space of other OWL ontologies and related to them via context mappings. Besana *et al* [Besana et al., 2005] present a framework for allowing agents to discover semantic mappings between contexts as these become necessary for communication. Potential bridges are hypothesised and confidence intervals assigned to them through a filtering process.

2.2.3 Relevance to the Project

- **What the project requires from ontologies:**

  It is necessary to base our concept of ontology on accepted definitions of ontology within the field, so as to ensure that our work is compatible with other

work being done with ontologies. Naturally, we cannot claim to have made a contribution to ontology refinement and matching if our concept of ontology is completely different to that which is generally used, and thus closely studying this field was essential to our work. The project particularly makes use of the KIF representation, and we have used the Ontolingua server to produce our ontologies.

- **What the project contributes to ontologies:**

  We do not consider that the project contributes directly to the theory of ontologies. The contribution of this project is concerned with the subfield of ontology refinement and matching; this subject is covered in detail in Chapter 3.

2.3 The Semantic Web

The development of the World Wide Web has caused an enormous change in global communication and the impact of computers on people's lives. However, the role of computers within the Web is to act as tools, giving access to the information but not processing or evaluating it. The WWW was designed for human comprehension and relies heavily on natural language and visual aids such as pictures, fonts and colours, which can be difficult for machines to analyse. But as the importance of web-based services has increased [McIlraith et al., 2001], it has become increasingly apparent that there would be enormous benefit in enabling machines to use the web and perform web-based tasks currently performed by humans [Narayanan and McIlraith, 2002]. The two main motivators of the Semantic Web are data integration and providing more intelligent support for end users [van Harmelen, 2004]. Thus much current research is focused on how to make the web structure and web-based services accessible to automated systems.

The World Wide Web Consortium (W3C), which is the organisation that is chiefly behind the development of the Semantic Web, define the Semantic Web as providing *"a common framework that allows data to be shared and reused across application, enterprise, and community boundaries"* [W3C, 2005].

In order to facilitate this, markup languages, which have well-defined semantics to enable unambiguous computer interpretation, are essential to the Semantic Web.

Just as the World Wide Web can viewed as a network for providing access to online services, the purpose of the Semantic Web is often viewed as facilitating services, that are provided, accessed and used by automated systems. The Semantic Web Services Initiative (SWSI) is an *ad hoc* initiative of academic and industrial researchers which aims to create infrastructure to support *"maximal automation and dynamism in all aspects of Web service provision and use"* [SWSI, 2005].

A key concept of the semantic web is the automated processing of ontological knowledge by agents and services. [Berners-Lee et al., 2001] outlines how these are central to the role of the semantic web, and how they require ontological matching to perform appropriately. However, the nature of the semantic web means that strict controls on these ontologies are impractical: it must allow for, amongst other things, partial information, erroneous information and the evolution of ontologies [Koivunen and Miller, 2001]. All of these requirements cause difficulties for agent communication, and create the need for online ontology mapping and refinement.

2.3.1 Ontologies on the Semantic Web

Since the central purpose of the Semantic Web is to allow semantic interpretation of remote information and services by automated systems and agents, the problem of how to represent this information to facilitate this is crucial. The meaning of everything on the Semantic Web has to be explicit, has to be available to systems or agents attempting

to access it, and has to be formally represented so that it is unambiguous. Thus ontologies are widely regarded as one of the foundational technologies for the Semantic Web [Antoniou et al., 2005].

Over the past few years, several markup languages have been developed which can make the contents of Web pages apparent to agents. The first of these is XML (Extensible Markup Language). Although this considerably improved the situation, it lacks semantics [McIlraith et al., 2001], hence ambiguity is possible and agents cannot be guaranteed to determine the intended interpretation of XML flags [SemanticWeb.org, 2002].

W3C developed RDF (Resource Description Framework) [Lassila and Swick, 1999] and RDFS (RDF Schema), which is an object-oriented type system which can be thought of as a minimal ontology language [McIlraith et al., 2001]. RDF Schema allows the following features to be expressed [Antoniou et al., 2005]:

- Classes and their instances;
- Binary properties between objects;
- Organisation of classes and properties in hierarchies;
- Types for properties: domain and range restrictions.

These are an improvement on XML but are still not expressive enough and have poorly defined semantics. RDF and RDFS are based on an unusual thesis of representation that is different from that built into most representation languages, and this makes its use as the foundation of representation in the Semantic Web difficult [Horrocks and Patel-Schneider, 2003].

In order to attempt to rectify the problems associated with RDF and RDFS, the DAML (DARPA Agent Markup Language) family was developed, and these are becoming the

web standard. OIL (Ontology Inference Layer) is an advanced ontology language that was designed to meet three requirements [Fensel et al., 2001]:

- to be highly intuitive to the human user;
- to have well-defined formal semantics with established reasoning properties;
- to have a proper link with existing Web languages such as XML and RDF to ensure interoperability.

The combination of DAML with OIL to create DAML+OIL [DAML+OIL, 2002], which is built on top of RDF(S) and extends its expressive power, was an important development. It was built from the original DAML ontology language DAML-ONT in an effort to combine many of the language components of OIL. The language has a clean and well defined semantics and a clearly specified syntax. DAML+OIL was carefully designed so that reasoning was decidable [Horrocks and Patel-Schneider, 2003]. DAML+OIL was used by W3C to create the OWL Web Ontology Language. DAML+OIL and OWL extend RDFS.

An important idea for representation on the Semantic Web is the idea of layering languages. This allows for a simple core that can accommodate simple taxonomies and relationships, while additional layers of expressivity, functionality, and complexity can be added for groups requiring more expressive power [Clark and Uschold, 2002]. In order to meet the different requirements that different users have for the Web, OWL is layered. The lowest layer is RDFS, on which all the OWL layers are based. The OWL layers on top of RDFS, in order of increasing complexity, are: OWL-Lite, OWL-DL and OWL Full. This approach enables users to use the maximal expressivity they require without dealing with the complications of expressivity they do not need.

However, like all DAML languages, OWL is largely based on Description Logic, and its semantics are thus very different from those of RDFS, making layering DAML+OIL

or OWL on top of RDFS problematic. A solution to this layering problem has been suggested through the development of languages such as SKIF [Hayes and Menzel, 2001] and L_{BASE} [Guha and Hayes, 2003], which have semantics that are more similar to those of RDFS. The core of SKIF and L_{BASE} is standard first order logic syntax, extended with the ability to use predicates and variables interchangeably [Horrocks and Patel-Schneider, 2003]. Horrocks and Patel-Schneider argue that it would in many ways be better not to use RDF and RDFS as the basis for Semantic Web ontologies, but instead to base them in first-order logic [Horrocks and Patel-Schneider, 2003]. However, using any language that is not based on RDF and RDFS could be considered to be a disadvantage, because it could not take advantage of the XML- and RDF-based technology that has allowed DAML+OIL to develop so quickly [van Harmelen, 2002].

Ontology Service Languages

The concept of services is centrally important to the Semantic Web: many common interactions on the Semantic Web are assumed to consist of agents performing services for one another. It is thus important that ontology languages that are used by agents on the Semantic Web are able to describe interfaces to services. This essentially involves being able to describe the conditions under which a service is performable and how things are changed after the service has been performed. The process can be thought of as analogous to an action description. However, it is difficult in most ontology languages to describe services. A simple definition for a service might include a list of conditions under which it was performable and a list of conditions that apply after it has been performed: in effect, preconditions and effects. However, a general description of preconditions and effects requires the use of variables, and ontology languages based on Description Logics do not include variables. Thus it is rather more complicated to describe services in such ontologies.

An attempt to provide this ability for DAML languages created DAML-S, a DAML-

based Web Service Ontology [DAML-S, 2002]. This was later updated to OWL-S [Services, 2005], which is compatible with OWL. It supplies web service providers with a core set of markup language constructs for describing the properties and capabilities of their web services in unambiguous, computer-interpretable form. OWL-S markup is designed to make web service tasks easier to automate. An OWL-S ontology is made up of three parts, as illustrated in Figure 2.1:

- The *Service Profile*: this is for advertising and discovering services, and describes what the service requires of the user and what it provides for them;
- The *Service Model*: this gives a detailed description of a service's operation, and describes how it works and what happens when the service is carried out;
- The *Service Grounding*: this provides details of how to interoperate with a service, via a message, and explains how the service is used: the necessary communication protocol, message formats and other service-specific details.

The service profile provides the information needed for an agent to discover a service and, taken together, the service model and service grounding provide enough information for an agent to make use of a service.

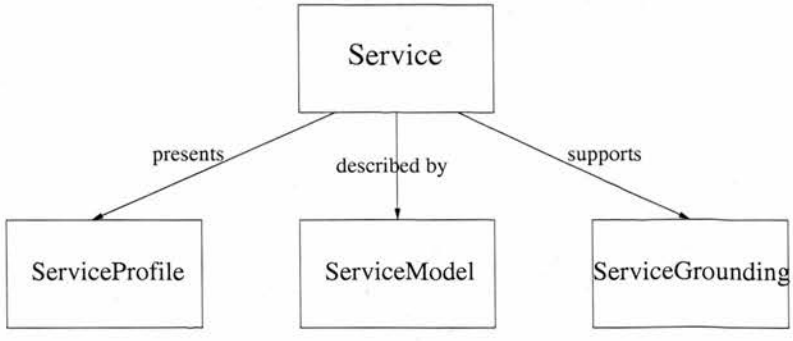

Figure 2.1: OWL-S Service

Similar to OWL-S is WSMO (Web Service Modelling Ontology) [WSMO, 2005], an initiative to describe web services using F-Logic [Kifer and Lausen, 1989].

2.3.2 Relevance to the Project

- **What the project requires from the Semantic Web:**

  The project does not directly build on the work done in this field. However, it is nevertheless important to the development of the project, because the Semantic Web, and other large-scale, multi-user, distributed systems provide the motivation for this work and the area in which it will be useful. The fact that the Semantic Web is an area of increasing import in Artificial Intelligence demonstrates that the scenario in which our work can operate is one that is of much importance.

- **What the project can contribute to the Semantic Web:**

  The main focus of the project could be considered to be aimed at systems like the Semantic Web, in which large numbers of disparate processes, or agents, are attempting to interact successfully. Systems such as this cannot always assume that these agents are identical, and yet differences between the agents present huge problems in their interactions, which are often difficult or impossible due to these differences. This project addresses one aspect of this problem: ontological mismatch. Since it is not possible to ensure mismatches do not exist, nor is it easy for agents to interact when mismatches exist, patching these mismatches as and when it is necessary is an important part of making systems such as the Semantic Web successful, and this project presents one approach to this problem.

2.4 e-Science and the Grid

According to the National e-Science Centre [NESC, 2005], e-Science is *"the large scale sciences that will increasingly be carried out through distributed global collaborations enabled by the Internet."* They state that this will require access to very large

data collections, very large scale computing resources and high performance visualisation for individual user scientists.

The infrastructure required for this is much more complex than is required for, for example, the World Wide Web, because scientists need access not just to pages of information but to remote facilities and resources and to dedicated databases. The focus is on allowing scientists to access instruments remotely and to share and combine their facilities and their results. In order to facilitate this, the architecture of the Grid is being developed.

According to Ian Foster and Carl Kesselman, the Grid is *"an infrastructure that enables flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions and resources."* [RCUK, 2004]. Foster and Kesselman are the inventors of the Globus approach to the Grid. Their Globus Toolkit is an open source software toolkit which is used for building most Grid systems and applications.

Like the Semantic Web, the Grid has to deal with the problem of multiple users who wish to be able to interact automatically. e-Science is designed as a tool for scientists, and it is imperative that it should not be difficult to use or place too many demands on users, such as insisting on them altering their representations to a common representation; if it does, scientists will prefer to rely on older techniques of communication and interaction and will not use the Grid. Thus the ability to deal with many kinds of mismatches is crucial to the success of the Grid and of e-Science.

2.4.1 Relevance to the Project

Since the Grid is also a large-scale, multi-user, distributed system, albeit a less tightly constrained one, the issues discussed in Section 2.3.2 are also relevant here.

2.5 Peer-to-Peer Systems

Peer-to-Peer computing is an area that has grown to prominence in recent years, and is the basis of many of the Internet's most important and innovative applications. It seems clear that the development of the Semantic Web will lead to an even greater dependence on peer-to-peer systems and a need to develop their functionality even further.

There is no definitive formalisation of what a peer-to-peer system is, but their essential characteristic is that they are distributed systems, with no central control or hierarchical organisation [Haase et al., 2004]. Each peer in the network is autonomous and has equal status with every other peer. Characteristics generally associated with peer-to-peer networks are [Oriol, 2003]:

- **Decentralisation**: there is no central control or management;
- **Potentially large size**: it is possible for a large number of peers to join;
- **Dynamism**: there are no constraints on peers' behaviour and abilities, and thus it is impossible to know when a specific peer may leave or join the network;
- **Heterogeneity**: the peers in the network may be very different;
- **Service provision**: peers may use the services of other peers and themselves provide services that are available to other peers.

The main advantages of peer-to-peer networks is that they remove the bottlenecks associated with centralisation, and therefore they allow efficient use of network resources. Such systems are more scalable and more robust than centralised systems. The potential search problems in peer-to-peer systems can be dealt with by a variety of techniques, such as distributed hash-tables (DHTs), or by semantic overlay networks

(SONs), or by some combination of the two. DHTs allow for storing and allocating content in a completely distributed way, by hashing each shared item to a unique key, which is efficiently routed to the unique peer responsible for that key. SONs enable peers to keep pointers to other peers which have similar content to them; thus queries in peer-to-peer networks can be more directed, and do not need to be distributed to every peer in the network [Haase et al., 2004]. pNear [Siebes, 2005] combines these two approaches and thereby circumvents some important disadvantages that hold for the individual approaches.

Some of the important applications that are based on peer-to-peer networks are [Mauthe and Hutchison, 2003]:

- **Napster**: which is an online music sharing system, the first such system based on peer-to-peer technology. It is, in fact, only partially peer-to-peer: the actual exchange of content is between peers, but the discovery of peers is centralised;

- **Gnutella**: which is another online file sharing network used primarily to exchange music, films and software. It is fully peer-to-peer; however, it suffers from problems of scalability and from free riders (peers who take content from other peers but do not provide any themselves);

- **MojoNation**: which is an online file store which is fully peer-to-peer and includes disincentives for free riders through a credit system;

- **Freenet**: which is a file/content sharing system where the primary goal is to make its use censorship-resistant by ensuring completely anonymity and preventing file deletion;

- **Bibster**: which is an application of the use of semantics in peer-to-peer systems, and enables exchanging bibliographic metadata amongst researchers.

Peer-to-peer techniques are not only useful on the Internet; they can also be used effectively in domains such as the Semantic Web, Grid computing, database systems and multiagent systems. In fact, the arguments for peer-to-peer technology in such environments is often more persuasive than on the Internet. For example, on the Semantic Web, there is not only a potential problem with physical bottlenecks, as on the Internet, but also with semantic bottlenecks, where semantics are enforced by a central server [Stuckenschmidt et al., 2005]. Making use of peer-to-peer technology will avoid both of these problems.

However, useful as peer-to-peer systems are, they are still not realising the full potential of peer-to-peer technology. One problem is that the knowledge structure of peers in peer-to-peer networks is generally quite limited. Peer-to-peer systems mostly rely on keyword search and are not appropriate for the requirements of more sophisticated knowledge structures, such as knowledge management systems [Ehrig et al., 2003]. The SWAP (Semantic Web and Peer-to-Peer) project integrates these two approaches, allowing participants to maintain individual knowledge structures whilst sharing knowledge in ways such that administration efforts are low but knowledge sharing and finding is easy. Another approach is to take an agent-based view of peer-to-peer [Singh, 2002], with agents acting as peers and modelling, communicating and learning about each other. Peer-to-peer systems can also be used for effective composition of web services [Walton, 2005] if the peers in a peer-to-peer architecture are represented and controlled through a web service interface.

2.5.1 Relevance to the Project

- **What the project requires from peer-to-peer:**

  The philosophy behind peer-to-peer technology informs the philosophy behind ORS, and we ultimately envisage the technology behind ORS being used in a peer-to-peer environment where one cannot make assumptions about other

agents and one must attempt to interact successfully with other agents despite heterogeneity and despite the lack of any central resources for overcoming these difficulties. However, since ORS is currently a proof-of-concept system, its inherent limitations mean that it is not currently peer-to-peer, nor have we taken many practical ideas from the field in the developing of ORS.

- **What the project can contribute to peer-to-peer:**

  One of the difficulties of peer-to-peer agent systems is that the agents must be able to understand one another, and yet the nature of peer-to-peer systems means that it is not desirable or possible to place too many conditions on peers. Investigating how peers might communicate without being entirely compatible is thus important, and ORS tackles one aspect of this: namely how agents with similar but misaligned ontologies can successfully communicate. The ideas of ORS could thus be useful in a peer-to-peer context, particular in a situation where the peers are agents.

2.6 Planning

The ability to form plans to achieve a goal and then to execute these plans in a given environment is generally considered to be one of the most fundamental and important aspects of intelligence. The field of planning has been significant since the earliest days of AI, and has progressed impressively over the years. However, there are still many problems that remain unsolved, and many ways in which automated planning is vastly inferior to human planning. In this section, we explain the development of planning throughout its history, the most significant fields today, and the relevance of planning to this project.

The field of automated planning grew out of AI problem solving in the early 1970s. Among the first significant planning system were QA3, developed by Cordell Green in

1969 and STRIPS (Stanford Research Institute Problem Solver), [Fikes and Nilsson, 1971], developed in 1971 by Fikes and Nilsson. At this time, as the name STRIPS suggests, planning and problem solving were considered analogous, though they have since diverged. QA3 represents sentences and questions in a first-order language and carries out deduction by resolution. STRIPS was used to drive the SHAKEY robot and addressed the problem of efficiently representing and implementing the operations of a problem. STRIPS has had a strong influence on the development of the planning field and even today a great many planners use a STRIPS-style representation.

STRIPS-style systems use operators which consist of three lists: *preconditions*, *add-list* and *delete-list*. The first list states what preconditions must be true before the operator can be applied, the second list states what will become true as a result of the application of the operator and the third list states things that were previously true and have now been made false. The representation assumes that all actions are instantaneous and that there is no interval between actions, which is not always a reasonable assumption. STRIPS makes use of triangle tables, which is a data structure for organising sequences of actions within a plan [Luger and Stubblefield, 1997]. The main advantage of triangle tables is that they offer some assistance if the plan breaks down due to an unexpected event. When something goes wrong, the planner can look into the rows and columns of the triangle table to find what is still true, and use this to work out what the next step should be to get the larger solution restarted.

Modern planning seems to fall naturally into two distinct threads: heuristic planning and planning graph based planning.

- **Heuristic Planning:** As the name implies, heuristic planning uses heuristics to guide the search process. One of the biggest challenges in planning is developing systems that can produce plans quickly. The reason why planning can be a very slow process is that the search spaces tend to be enormous. Using heuristics to reduce the search space can allow the production of planners that are

extremely time efficient. One of the most influential representatives of this direction if Hoffmann's FF planner [Hoffmann, 2003, Hoffmann, 2002, Hoffmann and Nebel, 2001]. The success of FF lies in the fact that it uses a relaxed problem to compute the heuristic for the search. More recently, another big advancement in heuristic search planning was achieved by LPG [Gerevini et al., 2003].

- **Planning Graph based planning:** Another popular planning paradigm was introduced with the Graphplan planner [Blum and Furst, 1995]. Graphplan is a partial order, least commitment planner, which is optimal with respect to the number of execution steps. The ideas behind graph plan are pivotal to the success of many of today's fastest planners, such as BlackBox [Kautz and Selman, 1999] and SATPLAN04 [Kautz and Selman, 1992].

Hierarchical planning is used in many planners, though not in STRIPS, so that plans can be described at both a high level, providing a good overall view of the plan, and a low level, providing precise instructions of what to do [Russell and Norvig, 1995]. There are two main types of hierarchical planning:

- **Hierarchical decomposition:** an abstract, non-primitive operator can be decomposed into a complex network of steps.
- **Abstraction hierarchies:** a single operator can be planned at different levels of abstraction. At the primitive level, the operator has a full set of preconditions and effects; at higher levels, the planner ignores some of these details.

Planners that have to deal with large and complex domains will inevitably be working with incomplete and incorrect information. Even in a simple domain such as the blocks world, unexpected events can occur, such as a block falling off the block it was placed on, which would lead to an incorrect theory. Thus it is important to develop planning systems that can be flexible in their approach and continue to move towards a goal even

if the intended route is not possible. An early approach to this problem is the STRIPS triangle tables, as discussed above. Other approaches are:

- **Conditional planning:** all the possible outcomes are considered in advance and the plan is developed to give a range of different options depending on the outcome of an action. This is not very practical in any but the smallest of domains as the plan becomes extremely large and a lot of effort is necessary to consider every possible outcome. In fact, in most domains it is not possible to predict with certainty every possible outcome.

- **Execution monitoring:** the outcomes of actions are monitored to check whether they are as expected, and, if not, replanning is performed. Thus the job of considering what to do in the case of failure is deferred until it is necessary to consider it. This an advantage over conditional planning because it means there is no unnecessary work on parts of the plan that do perform as expected. However, there are also inherent problems in that it can produce fragile plans without the ability to fix themselves. In practise, a combination of the two is often used.

The above methods are ways to patch plans that fail to work due to unexpected events. Another problem is how to form plans with knowledge that is known to be inadequate. Two approaches to this are **coercion**, where one forces the world into a possibly inaccurate model and assumes that this will be sufficient; and **abstraction**, where one ignores detail as much as possible. These approaches allow plans to be formed from incomplete knowledge but run the risk of forming incorrect plans. It is not reasonable to assume that either approach is valid; however, if the knowledge is incorrect or incomplete then it is impossible to form reliable plans and it may be decided that a potentially incorrect plan, formed using one of the above approaches, is more useful than no plan at all.

2.6.1 Proof Planning

Proof planning is a meta-level reasoning technique developed at Edinburgh University for guiding the search of a proof [Bundy, 1991, Bundy, 1998]. Proof planning can be considered as plan formation in the traditional AI sense, although there are some differences, mostly to do with the fact that proof planning takes place in the deterministic domain of mathematics, whereas traditional planning deals with incomplete and uncertain information.

Proof planning splits the problem of finding a proof into two:

1. an approximate plan is assembled by the plan engine;
2. the plan is executed by the theorem prover.

A proof plan captures general knowledge about the commonality between the members of a proof family and is used to guide the search for more proofs in that family. Proof planning with critics involves the combination of a proof planner and an exception handler. The exception handler is invoked if the goal is blocked. A proof critic contains a high-level specification of a failure pattern as well as providing the corrective action. Proof planning is cheaper than searching for a proof in the underlying theory because each plan step covers many proof steps and also because plans contain many heuristics.

2.6.2 Relevance to the Project

- **What the project requires from planning:**

  Making use of automated planning techniques is essential to this project. The overall aim of what we are doing certainly has uses and importance outside the realm of planning; however, this is the domain in which have chosen to explore the problem. The planner is an essential component of the system, without which

the system could not operate, but the theoretical requirements of the planner are fairly simple, and we do not require much of the sophistication of modern planning techniques.

- **What the project can contribute to planning:**

  Although the central aims of the project are tangential to the planning community, the process of developing a planning system that deals with ontologies has allowed us to make some contributions to the crossover areas between planning and ontology which, with the development of the Semantic Web and similar multi-user distributed systems, is becoming increasingly important. These include the notion of a *plan deconstructor* and a translation process between KIF and PDDL. These are discussed in more detail in Chapter 4 and in Sections 6.3 and 9.1.2. Additionally, critics that repair a signature have not previously been considered in proof planning, but could provide useful additional functionality. However, we have not investigated this possibility in any detail.

2.7 Summary

This chapter introduces the theory on which ORS depends, describes in brief the relevant fields, and explains how our work both takes from and adds to these fields.

Chapter 3

Mismatch and Refinement

The focus of this project centres on facilitating the interaction of agents that have mismatched ontologies by finding ways to refine them appropriately. There has been much work in AI on the problems of refinement and revision in various different contexts, and on the problem of how to maintain consistency between or within knowledge bases. The most important approaches are discussed in this chapter. We briefly examine each approach, and then discuss the subject in the context of our work. A further discussion of the approaches in this chapter that most closely relate to our work, together with a detailed analysis of how they compare, can be found in Section 7.5.

3.1 Ontology Mismatch

3.1.1 Introduction

The notion of ontology, as discussed in Section 2.2, was originally concerned with defining the existence of things in the world. From this point of view, an ontology can be considered to be objective and absolute: disagreement over the contents of an

ontology is not permissible. However, such a view is clearly unworkable in an environment where ontologies are interpreted, updated and changed. The emergence of the Semantic Web, with its distributed and deregulated nature, has exacerbated this problem [Kalfoglou and Schorlemmer, 2004], but the problem of integrity of knowledge bases is far older than this. Even in the field of databases, which are much more controllable than the Semantic Web because they can be regulated and are not distributed, this issue has to be faced [Banerjee et al., 1987]. Additionally, the field of belief revision ([Belief Revision, 2005]; see Section 3.2) is concerned with how a system of beliefs can be maintained under conflicting information. In fact, [Finkelstein et al., 1993] makes it clear that semantic heterogeneity is something that is an inevitability in distributed systems.

The central role of the Semantic Web can be viewed as facilitating agents from different backgrounds and perspectives to interact successfully. Differences between ontologies that interacting agents may have access to threaten their ability to interoperate. ISO/IEC 2382 defines interoperability as *"the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units"* [Kalfoglou and Schorlemmer, 2004]. In a multi-agent context, Uschold argues that *"two agents are semantically integrated if they can successfully communicate with each other"* and that *"successful exchange of information means that the agents understand each other and there is guaranteed accuracy"* [Uschold, 2002]. The idea that guaranteed accuracy is essential is not universally accepted; it is becoming increasingly common to define a notion of "good enough" [Kalfoglou and Schorlemmer, 2004], and ensuring that agent communication complies with this standard, rather than guaranteed accuracy. This notion has developed pragmatically, because insisting on perfection in a system such as the Semantic Web is unworkable. This mirrors the situation in human communication where, due to the complexity of interaction, misunderstandings are always possible.

3.1.2 How and Why Ontologies Differ

As discussed in Section 2.2, the term ontology does not always mean the same thing to all people. Determining what we mean by ontology mismatch and how this can occur is obviously dependent on what we mean by ontology. In this project, we follow the definition of ontology expounded by Kalfoglou and Schorlemmer, which can be found in Section 2.2, where an ontology consists of both a signature and a theory, though some people define ontology to be only the signature. Kalfoglou and Schorlemmer explain that a basic approach to the problem of mismatched ontologies is to consider only morphism of the ontological signature, whereas a more ambitious and practically necessary approach is to consider how ontological axioms are mapped as well as the signature [Kalfoglou and Schorlemmer, 2003b]. We, like most involved in resolving ontology mismatch, are centrally concerned with dealing with signature mismatch, but, like Kalfoglou and Schorlemmer, agree that dealing with the axioms (or theory) is a practical necessity in order to maintain internal consistency within an ontology.

A very common approach to ontologies is to view them as a taxonomy of concepts, or a class hierarchy [Giunchiglia and Shvaiko, 2003, Giunchiglia et al., 2005a, Doan et al., 2003,Campbell and Shapiro, 1998,Kalfoglou and Schorlemmer, 2003a,Wiesman et al., 2002]. Thus ontology mismatch is concerned with differences in the taxonomy of ontologies, and finding links between the concepts.

Hameed *et al* [Hameed et al., 2003] list three reasons why there are differences between ontologies:

1. There are different models to choose from and, even if some become standard, there will still be legacy differences;
2. Ontologies are often aligned with particular perspectives on the world;
3. Ontologies are evolving.

It is clear that these are all not only inevitable but desirable features of ontologies, and that the use of ontologies must develop to be able to deal with them.

3.1.3 Resolving Ontological Differences

The literature discusses four main approaches to the problem of mismatched ontologies: **mapping**, **merging**, **aligning** and **translation** [Kalfoglou and Schorlemmer, 2003b]. These differ in the degree to which they attempt to resolve the differences between the ontologies and in whether they produce a single ontology or an intelligent way to interpret each ontology in terms of the other.

3.1.3.1 Ontology Mapping

The process of ontological mapping attempts to build maps between the concepts found in the two ontologies. According to Kalfoglou and Schorlemmer, ontology mapping is *"the task of relating the vocabulary of two ontologies that share the same domain of discourse in such a way that the mathematical structure of ontological signatures and their intended interpretations, as specified by the ontological axioms, are respected"* [Kalfoglou and Schorlemmer, 2003b]. According to Stuckenschmidt [Stuckenschmidt, 2002], ontology mapping preserves the autonomy of ontological models on a general level because agents are able to keep their own ontologies and not conform to a single merged version. He also claims that the mapping approach to dealing with mismatched ontologies is the most often mentioned in the literature.

Ontology mapping is difficult. This is, for example, because [Kalfoglou and Schorlemmer, 2003a]:

- There are many implicit assumptions in ontologies;

- Systems that perform ontology mapping are embedded in an integrated environment for ontology editing or are attached to a specific formalism;
- Mapping and merging are usually based on heuristics that mostly use syntactic clues to determine correspondence or equivalence between ontology concepts, but rarely use semantics;
- Most, if not all, approaches do not treat ontological axioms or rules often found in formal ontologies;
- Ontology mapping is a term used in different ways by different people.

Giunchiglia *et al* [Giunchiglia et al., 2005a,Giunchiglia and Shvaiko, 2003,Giunchiglia et al., 2005b] reiterate the problem of determining mappings purely through syntactic means, rather than also using semantics. They attempt to resolve this problem through their use of *semantic matching*, which is implemented in their system S-Match, which is discussed in more detail in Section 3.1.4.

Wache *et al* [Wache et al., 2001] list four common ways of facilitating ontology mapping:

1. Through defined mappings, which can be predefined or determined during interaction through mediation agents (see Section 3.1.5);
2. Through lexical relations, which provide intuitive semantics for mapping between concepts in different ontologies;
3. Through top-level grounding, where a single top-level ontology can be used to define mappings;
4. Through semantic correspondences, where well-founded semantic correspondence between concepts from different ontologies can be identified.

Hameed *et al* [Hameed et al., 2003] describe different levels at which this mapping can take place:

- Pairwise mappings between different individual ontologies - this may have to be done many times, once for each pair of ontologies;
- Mapping to a single common ontology, which is more scalable but removes flexibility;
- Mapping between multiple ontology clusters, with inter-cluster mappings, which, they claim, is the best approach as it combines the advantages of the other two approaches.

Stuckenschmidt believes that ultimately, pre-existing mappings will rarely exist between the networks of ontologies that exist on the Semantic Web because it would be too expensive to construct them. Rather, these mappings will mostly be established by individual agents that are using these different ontologies, in order to process a given task [Stuckenschmidt, 2002].

3.1.3.2 Ontology Merging

Ontology merging is a rather different approach to resolving conflict between ontologies. Unlike ontology mapping, which maintains distinct ontologies but determines ways to relate them, ontology merging results in a single ontology: two different ontologies are scrapped and instead a new one is used that to some extent covers both of them.

Hameed *et al* define merging as *"the act of building a new ontology by unifying several ontologies into a single one"* [Hameed et al., 2003], and Noy and Musen view merging as *"a single coherent ontology that includes the information from all sources"*.

Ontology merging is less common than ontology mapping, and can be problematic because the autonomy of agents is partially lost by the use of a global ontology [Stuckenschmidt, 2002]. It is also more difficult than mapping because of the construction of this single ontology. Ontology mapping can be thought of as only a fragment of the more ambitious tasks of merging and aligning ontologies [Kalfoglou and Schorlemmer, 2003b].

Mitra and Wiederhold claim that a merging approach of creating a unified source is not scalable; one monolithic information source is not feasible due to unresolvable inconsistencies between them that are irrelevant to the application [Mitra et al., 2000].

3.1.3.3 Ontology Alignment

Ontology alignment is a more complete form of mapping, and involves mapping concepts and relations to indicate equivalence [Kalfoglou and Schorlemmer, 2004].

Noy and Musen consider alignment to be a process in which *"the sources must be made consistent and coherent with one another but kept separately"* [Noy and Musen, 2000].

3.1.3.4 Ontology Translation

Translation can be considered to be an implementation of the mapping process. Whilst ontology mapping defines a collection of functions that specify which concepts and relations correspond to which other concepts and relations, ontology translation is the application of these mapping functions to the sentences based on one ontology into the other [Kalfoglou and Schorlemmer, 2003b].

3.1.3.5 Mediators and Brokers

Campbell and Shapiro have developed the Ontological Mediator for resolving communication difficulties resulting from different ontologies [Campbell and Shapiro, 1998]. An ontological mediator is an agent capable of reasoning about the ontologies of two communication agents, A_1 and A_2. If agent A_1 uses word W, the mediator must look for an ontological translation W' that means the same thing to A_2. It is assumed that common words mean the same to both agents. Mediators are not concerned with the process of detecting misunderstandings, but rather with ways of resolving communication problems. In Campbell and Shapiro's definition, ontologies consist of concepts which are represented by words; one or more words per concept. Words can be shared between agents but concepts cannot.

This is closely related to the idea of brokers, where *broker agents* facilitate other agents communicating with each other. For example, the AKT project [AKT, 2002] has produced a brokering system through which agents are able to advertise their capabilities or to look for other agents to perform tasks for them [Schorlemmer et al., 2002, Sleeman et al., 2002]. If the ontologies of the agents are not compatible then *bridges* are defined to translate terms in one ontology to be described according to the other ontology. These bridges are usually constructed manually, hence this is not a feasible solution for ontologies that are widely different. Additionally, they can only be defined if the ontology of a knowledge system is made explicit. However, they are a powerful concept for extending the range of the knowledge and capabilities of any system.

3.1.3.6 Ignoring Mismatches

Instead of attempting to diagnose and repair inconsistency, another approach is to simply avoid the inconsistency. Reasoning with inconsistent knowledge bases is problematic due to the *ex contradictione quodlibet* premise, which states that from contra-

diction, anything can be deduced. However, if the classical logic approach is ignored in favour of a non-standard approach such as paraconsistent logic, then meaningful reasoning becomes possible. Huang *et al* [Huang et al., 2005] discuss how selection functions can be used to select some consistent sub-theory from an inconsistent ontology, so that standard reasoning techniques can then be used to find sensible answers. They introduce their system PION that implements this approach. However, this deals with the problem of internal inconsistency, rather than inconsistency between agents.

3.1.4 Systems for Resolving Ontological Differences

In this section, we describe some of the systems that perform these tasks. This is not intended as a comprehensive survey, but rather to give an overview of the kind of functionality that is available.

Noy and Musen developed PROMPT [Noy and Musen, 2000], and later PROMPTDIFF [Noy and Musen, 2002], which use linguistic similarity matches between concepts and the underlying ontological structures of the environment of the open-source ontology editor Protégé-2000, for which they are available as plug-ins. PROMPT offers semi-automatic, interactive ontology-merging, which guides users through the merging process, determining conflicts and making suggestions and proposing resolution strategies through syntactic analysis. It is highly interactive and relies on the candidate ontologies already having close similarity. Anchor-PROMPT [Noy and Musen, 2001] implements a number of extensions, including consideration of similarity between class hierarchies [Hameed et al., 2003].

The use of information flow methods for mapping ontologies has been implemented in IFF by Kent [Kent, 2000] and in IF-MAP by Kalfoglou and Schorlemmer [Kalfoglou and Schorlemmer, 2003a]. IFF represents ontologies as logics, and ontology sharing as a specifiable ontology extension hierarchy. IF-MAP is based on the Barwise-Seligman

theory of information flow, which provides a systematic and mechanised way for deploying channel theory, a mathematical theory of information flow, on a distributed environment to perform ontology mapping among a variety of different ontologies.

GLUE [Doan et al., 2003] is an interactive system that uses machine learning to find semantic mappings between ontologies. It uses multiple learning strategies, each of which exploits well a different type of information either in the data instances or in the taxonomic structure of the ontologies. GLUE regards concepts as labels and recasts the problem as finding the best label assignment to concepts, taking into account both given knowledge about the domain and the two taxonomies and domain independent considerations such as neighbourhood. CGLUE extends GLUE by finding complex mappings between two given taxonomies. The authors claim this is an advantage over most matching systems, which deal only with simple mappings.

ONION (ONtology CompositION) [Mitra et al., 2000] proposes a scalable and maintainable approach based on interoperation of ontologies to handle distributed queries crossing the boundaries of the underlying information systems. This requires the interoperation between the ontologies for the individual information systems to be precisely defined. ONION matches two sets of concepts using dictionaries and information retrieval techniques.

Chimera [McGuinness et al., 2000] facilitates the merging and testing of ontologies. McGuinness *et al* consider the task of merging two ontologies into one or combining two or more ontologies that may use different vocabularies and may have overlapping content. Users can request analysis or guidance during the merging process. Help offered during the merging process includes: generation of a name resolution list that helps the user in the merging task by suggesting terms, each of which is from a different ontology, that are candidates to be merged; and generation of a taxonomy resolution list where the tools suggest taxonomy areas that are candidates for reorganisation [Hameed et al., 2003].

OntoView [Klein, 2002] compares ontologies at a structural level, showing which definitions of ontological concepts or properties have changed. It identifies changes among different ontology versions.

S-Match [Giunchiglia et al., 2005a, Giunchiglia and Shvaiko, 2003, Giunchiglia et al., 2005b] is a system based on *semantic matching*, a notion devised by Giunchiglia *et al* in which the key intuition is to exploit the model-theoretic information which is codified in the nodes and the structure of the graph. Whereas most approaches deal only with syntactic similarity measures and search for semantic correspondences only on the basis of syntactic features, semantic matching searches for semantic correspondence by considering not only the labels of nodes, but also their positions in the graph structure of the ontology. This approach, like many of the others, views ontologies as a concept taxonomy represented in a graph, and thus the position in the graph of a node (or concept) is crucial to its meaning. S-Match takes two trees and, for any pair of nodes from the two trees, computes the strongest semantic relation holding between the concepts of the two nodes. The kinds of semantic relations that are considered are: equivalence, more general, less general, mismatch and overlapping.

Stuckenschmidt [Stuckenschmidt, 2002] describes an approach in which agents share parts of their ontology. An agent will use one or more ontologies to perform a task. These ontologies will normally supplement each other but have sufficient overlap that an agent can find mappings between them, and each agent will establish mappings between ontologies which it is using. It is argued that on-demand translations are more appropriate in this situation than mapping and merging. The advantages of these two approaches are combined by defining the concepts from different sources using terms in a shared ontology and relating these concepts only if such a relation is needed during terminological reasoning. Connections to every other source in the system do not have to be established, only with the shared model. If a query cannot be fully understood, a rewriting is found that approximates it the most closely.

Ehrig and Sure [Ehrig and Sure, 2004] describe an approach that has been derived from the SWAP project, where the aim is to enable individuals to keep their own work views and at the same time share knowledge across a peer-to-peer network. Similarity is determined through rules which have been encoded by ontology experts, which are then combined for one overall result. Only one-to-one mappings between single entities are considered. Implementation is based on manually encoded mapping rules, which identify possible mappings.

SHOE (Simple HTML Ontology Extensions) [Heflin and Hendler, 2000] is a web-based knowledge representation language that supports multiple versions of ontologies. Ontology revision is defined as a change in the components of an ontology, so that it can involve the addition or removal of categories, relations and/or axioms. Revisions are only thought of as additions and removals; the modification of a component is thought of as a removal followed by an addition. SHOE performs versioning by maintaining different web pages for each version of an ontology. Each page must state which version it commits to and which ontologies it is compatible with.

C-OWL [Bouquet et al., 2004], mentioned in Section 2.2.2, also presents a solution to the problem of mismatched ontologies. Contextualised ontologies, which are local and not shared, are likely to contain many mismatches. C-OWL integrates these ontologies by facilitating the building of semantic bridges between the different ontologies in order that their concepts, roles and individuals can be related.

3.1.5 Determining Potential Mismatches

Another approach to this problem, particularly relevant to our work, is the idea of determining in advance what the potential mismatches are. For many of the methods described above, such an approach would be pointless, because they view ontological mismatch as a misalignment between concepts. However, if a richer view is taken of

ontologies, then the kinds of mismatches that are possible become more varied.

Visser *et al* [Visser et al., 1997] defined potential mismatches in the following way. In defining ontology mismatches, they confine themselves to mismatches between definitions of classes (CD), definitions of relations (RD) and definitions of instances (ID), and do not address definitions of functions or axioms. Definitions are $Def = < T, D, C >$, in which T is the term that is being defined, D is what it is defined in terms of, and C is the ontology-concept description to be defined, in natural language. They then provide a classification for potential mismatches, which, they emphasise, is not complete or disjoint but rather a useful instrument to determine which mismatches are hard to resolve, and to define guidelines for the design of interoperable systems. The classification is as follows:

- **Conceptualisation mismatches** - mismatches between two (or more) conceptualisations of a domain:
  - **Class mismatches** - mismatch concerning classes and their subclasses:
    - **Categorisation mismatch** - when two conceptualisations distinguish the same class but divide this class into different subclasses;
    - **Aggregation-level mismatch** - when two conceptualisations recognise the existence of a class, but define classes at different levels of abstraction.
  - **Relation mismatches** - mismatch relating to the relations distinguished in the conceptualisation:
    - **Structure mismatch** - when two conceptualisations distinguish the same set of classes but differ in the way these classes are structured by means of relations;
    - **Attribute-assignment mismatch** - when two conceptualisations differ in the way they assign an attribute (class) to other classes;

    * **Attribute-type mismatch** - when two conceptualisations distinguish the same attribute (class) but differ in their assumed instantiations.

- **Explication mismatches** - mismatches that are not defined on the conceptualisation of the domain but on the way the conceptualisation is specified. Since a definition has three components, there are eight different relations between two definitions. However, two of these situations do not require investigation: if all the terms are the same, there is no mismatch, and if all the terms are different, there is also no mismatch, because there is no similarity. The remaining six mismatches are:

  - **Concept mismatch (C)** - the concepts (natural language definitions) do not match;
  - **Definiens mismatch (D)** - the terms that are formally describing the definition do not match;
  - **Term mismatch (T)** - the terms that are being described do not match;
  - **Concept and Definiens mismatch (CD)** - neither the concepts nor the definiens match;
  - **Concept and Term mismatch (CT)** - neither the concept nor the term match;
  - **Term and Definiens mismatch (TD)** - neither the term nor the definiens match.

They discuss how mappings could be defined for these mismatches, and for which mismatches this would prove difficult.

Wiederhold [Wiederhold, 1997] classify potential mismatches thus:

- **Key difference** - different naming for the same concept (e.g., synonyms);

- **Scope difference** - distinct domains, or distinct coverage of domain members;
- **Abstraction grain** - varied granularity of detail among definitions;
- **Temporal basis** - mismatches concerning time;
- **Domain semantics** - distinct domains, and the way they are modelled;
- **Value semantics** - differences in the encoding of values.

Shaw and Gaines [Shaw and Gaines, 1989] deal with the problem of how to create a database of expert knowledge that has been obtained from many different experts, and thus may not be compatible. They describe four ways in which experts' ontologies might overlap:

- **Conflict** - experts use same term for different concepts;
- **Correspondence** - experts use different terms for same concept;
- **Contrast** - the experts use different terms and have different concepts;
- **Consensus** - experts use the same term for the same concept.

They discuss a methodology for facilitating experts coming to agreement. Automated tools are used to analyse the mismatches, but the mismatches are then resolved by the experts.

Hameed *et al* [Hameed et al., 2002] discuss how expert's ontologies can be used to create a single coherent ontology. Naturally, the experts will have different internal representations of their subject, and the ontologies they build will be mismatched in some ways. They compare the mismatches they discovered between expert's ontologies when these processes were carried out to mismatches defined by Visser [Visser et al., 1997], Wiederhold [Wiederhold, 1997] and Shaw and Gaines [Shaw and Gaines, 1989].

Stuckenschmidt and Klein [Stuckenschmidt and Klein, 2003] also take this approach. They are centrally concerned with ontology as a concept hierarchy, but consider the concepts as complex objects. Their ontologies are written in OWL-Lite, and they list all the possible change operations to an ontology according to the OWL-Lite knowledge model. This list includes both atomic change operations and complex change operations. 120 such changes are defined thus far, and the number is growing.

The approach of Stuckenschmidt and Klein was influenced by the work of Banerjee *et al* on schema evolution [Banerjee et al., 1987]. They developed ORION, a framework for supporting schema evolution. They established a taxonomy of over 20 schema changes, including creation and deletion of a class, alteration of the IS-A relationship between classes, and addition and deletion of instance variables and methods. Franconi *et al* [Franconi et al., 2000] add to the classical schema change primitives described in [Banerjee et al., 1987] by enabling the definition of complex and articulated schema changes. The potential changes that they developed include: add-attribute, drop-attribute, change-attribute-name, change-attribute-type, add-class, drop-class, change-class-name, change-class-type, add-is-a and drop-is-a.

3.1.6 Drawbacks and Limitations of Current Methods

This section demonstrates that there is a considerable amount of effort being spent on resolving the problems surrounding ontological mismatch. However, the difficult nature of the task means that there is as yet no complete solution to the problem, nor are any such solutions currently on the horizon.

Most of these approaches are interactive; it is far rarer to find a system that can perform such a task in a fully automated manner. It is preferable that any matching that is to be done on the Semantic Web during runtime of the interactions of agents should not be interactive; the agents should be able to resolve such problems themselves. If

the process of mapping/merging/aligning the ontologies is not done during runtime, then this problem is less pronounced. However, Kalfoglou and Schorlemmer claim that *"the advent of the semantic web, the proliferation of ontologies nowadays, and agent technology advances, pose hard requirements on the timescales for performing ontology mapping"*.

For most ontology mapping, it is necessary to translate to a common representation. A few systems do not use this method, but their processes are manual, laborious and presupposes that the knowledge engineer is familiar with the input formalisms, and does thorough inspection of the model semantics and domain to write meaningful mapping rules.

In summary, all such systems require large amounts of human effort to be put in the task in order to be able to map, merge or align ontologies sufficiently. This is problematic for the Semantic Web, where it is not possible for a user to predict which agents it will be necessary to interact with, how their ontology will clash with the user's agent's ontology, how many times ontologies will have to be altered/merged, and so on.

3.2 Belief Revision

Another field that is concerned with the integrity of knowledge bases is belief revision. Belief revision is centrally concerned with the problem of how to maintain the integrity of a set of beliefs about the world, given that these beliefs are constantly challenged by the environment. For an agent to act effectively within a domain, it is essential that it has beliefs about that domain. However, it may become apparent that a particular belief that it holds is, in fact, incorrect, perhaps because the domain has changed since it came to accept the belief, or perhaps it was wrong to originally accept the belief. The main focus of belief revision is on how the agent should restructure its belief system in the face of this contradiction of one of its beliefs [Gärdenfors, 1992]. It is thus

concerned with problems analogous to those of theory revision, and not with signature revision.

In some situations, the offending belief may exist in isolation in the belief system, so that it can be removed without concern for the integrity of the rest of the belief system. However, in most situations, the relationship of a belief with the other beliefs in the system is more complex. For example, beliefs are very often entailed by other beliefs, and if an entailed belief is discovered to be false, it is clear that at least one of the beliefs that entailed it must also be false.

In a closed belief system, any belief entailed by existing beliefs in the belief system must also be members of the belief system. Thus if a belief system contains the following two beliefs:

1. *All men are mortal*
2. *Socrates is a man*

then it is clear, by deduction, that the belief *Socrates is mortal* must also be in the belief system. If further interaction with the domain reveals that, in fact, Socrates is immortal, then not only must the belief *Socrates is mortal* be retracted, but so also must at least one of the premises of the deductive argument. Additionally, the impact on any beliefs that are entailed by the removed belief must be calculated. Often, this will mean that those entailed beliefs must also be removed; sometimes, they may be separately entailed by other beliefs in the belief system, and thus can remain.

Belief revision is concerned with how this should be done. Usually, it is desirable that this should be done with minimal impact on the belief system; that is, do not remove two beliefs if it is possible to maintain the integrity of the belief system by removing only one. However, in some situations, one belief may be considered more important to the belief set than two other beliefs, and thus these two would be removed in preference to that single one.

3.2.1 Representations of Belief States

Belief sets are often used as a way of representing a belief state. A belief set is defined as a set K of sentences in L which is closed under logical consequence. This is an appealing way of representing belief states because it means that if we have a belief A and we see that $\neg A$ is not in the belief set, we can conclude that adding A to the belief set will not cause inconsistency. However there are obviously computational problems with belief sets since they are usually infinite.

A more practical method of representing belief sets are belief bases, which are finite subsets of belief sets. If $Cn(A)$ is defined as the set of all the logical consequences of A, then we say H is a belief base of a belief set K if $Cn(H) = K$. Belief bases have another practical advantage over belief sets in that they allow the distinction between innate beliefs (*i.e.*, ones that we hold but can't justify) and justified beliefs (*i.e.*, ones that are justified by other beliefs in the database). However, checking for falsity ($\neg A$) is harder than in belief sets.

3.2.2 The AGM Postulates and Integrity Constraints

The AGM postulates were laid out by Alchourron, Gärdenfors and Makinson [Gärdenfors and Rott, 1995] as a framework for modelling and implementing belief revision systems. They are applicable to closed belief systems, and require that minimal change take place during belief revision.

The AGM postulates are based around expansion, revision and contraction. They depend on the assumption that, for any belief set K and sentence S in a language L, there is a *unique* belief set $K + S$ representing the revision of K with respect to S. This is a strong assumption.

In order to address the problem of choosing which beliefs to retract when logic alone

cannot help, these systems can make use of the notion of epistemic entrenchment. This is based on the idea that we may not consider all beliefs to be of equal value. For example, if a belief is fundamental to many other parts of our belief set and if it is in common use then we would give this a high degree of epistemic entrenchment and only retract it if forced to. A belief that is not of much use would receive a low degree of epistemic entrenchment. Hence each belief is annotated with a value that reflects its importance and the willingness of the system to renounce it. It may be considered better to retract several beliefs with low epistemic entrenchment than a single belief with a high epistemic entrenchment.

The AGM postulates require the fulfilment of the integrity constraints which are as follows:

1. Databases should be consistent, that is if A is in the database, $\neg A$ cannot also be in the database;

2. Elements logically entailed by the database should be in the database (closure);

3. Minimal Information loss: during a revision or contraction, the smallest possible amount of information should be lost that will allow the database to remain consistent;

4. Epistemic entrenchment is required.

Other methods of belief revision adhere to these constraints to varying degrees. They are certainly very useful computationally because they force a high degree of simplicity. However, this simplicity can be very limiting to the attempt to realistically model beliefs, and for this reason many areas of belief revision chose to disregard some or all of these constraints.

3.2.3 Foundation Theory v Coherence Theory

There are two main ways in which revision of the database can take place.

1. **Foundations Theory** - this states that the justifications of a belief should be tracked and nothing should be accepted in the database if it cannot be justified;
2. **Coherence Theory** - as long as a belief to be added is coherent with other beliefs in the database and doesn't entail any contradictions, then it can be accepted in the database without justification.

These are comparable with the principle of positive undermining and the principle of negative undermining, the first of which states that we should keep believing something until we are forced not to (inertia of belief), and the second of which states that we should stop believing something when we are not forced to.

3.2.4 Truth Maintenance Systems

A large group of belief revision systems are referred to as truth maintenance systems (TMSs). These systems focus on the justifications for the beliefs and manage beliefs on the basis of dependency information. They are connected to a problem solver which communicates new information to the TMS in the form of justifications.

One way in which TMSs differ from other forms of belief revision is that they never delete any beliefs. All beliefs are marked either *in* or *out* and, during updating, old *in* beliefs that are now rejected are marked as *out*, rather than being deleted. So updating becomes a process of relabelling nodes. TMSs are based on a severely restricted language; nodes are unstructured objects and justifications are one-way rules. They are capable of distinguishing between premises, which need no justification, and justified beliefs.

One of the main purposes of TMSs is to provide justifications or explanations for propositions. These are in the form of a list of the other propositions in the knowledge base that deductively entail the proposition. These can be in the form of assumptions, so that a justification would be of the form: *if proposition P is true and proposition Q is true then proposition R can be inferred.* The simplest form of TMS is the justification-based truth maintenance system, or JTMS. In a JTMS, each sentence is marked with one or more justifications that indicate why it is believed to be true. If, during retraction of beliefs, all of these justifications come to contain retracted beliefs, the sentence does not then have a valid justification, and is marked as *out*. The information about the justifications is retained, and if any of the retracted beliefs is reinstated, thus producing a valid justification, the sentence is then marked as *in*. More popular are ATMSs (Assumption-based Truth Maintenance Systems). These differ from JTMSs because they allow the representation of all states that have ever been considered, whereas JTMSs only allow one world state to be represented at a time.

Another advantage of TMSs is that they help in dealing with inconsistencies by helping to point out where the contradiction that an inconsistency entails lies. However, although TMSs can be extremely useful, reasoning with them is NP-hard.

3.3 Theory Change

Much work has been done in the more general field of theory change. By theory change, we mean exploring the problems surrounding mismatch and counter-examples that we have discussed throughout this section, but without the specific focus on ontologies or knowledge bases that are directly comparable to ontologies. In this section, we cover the some of the more interesting and relevant areas of this field.

Method	Description
Induction	Generalising from particulars
Surrender	Abandoning a conjecture in the light of counter-examples
Monster barring	Modifying a definition to exclude counter-examples
Monster adjusting	Reinterpreting a counter-example so that it complies with the theory
Exception barring: piecemeal exclusion	Considering the properties which unite all counter-examples and rewriting the theory definition to exclude them
Exception barring: strategic withdrawal	Withdrawing the theory (rewriting the definition) from examples for which it does not hold
Lemma Incorporation	Adding a precondition to the theory definition to exclude results for which it does not hold (making hidden assumptions explicit)

Table 3.1: Lakatos's Methods

3.3.1 Developing Mathematical Theories

Lakatos investigated the ways in which mathematical theories could account for counter-examples, and ways in which they could be altered so that counter-examples no longer provide a refutation of the theory [Lakatos, 1976]. The methods Lakatos describes are outlined in Table 3.3.1.

Pease *et al* [Pease et al., 2004] have implemented a multi-agent system on top of HR [Colton, 2002], which is a system for automated theory formation. The resulting system makes use of Lakatos's methods to fix faulty theories that may be generated within this multi-agent context.

Hayes-Roth [Hayes-Roth, 1983] presents five heuristic methods for repairing flawed beliefs which extend and make practical Lakatos's ideas of proofs and refutations. These are described in Table 3.3.1.

Method	Description
Retraction	Restricting the theory's predictions to be consistent with observations
Exclusion	Barring the theory from applying to the current situation cf Lakatos's monster-barring
Avoidance	Ruling out situations that predictably deny the theory's predictions
Assurance Method	Ruling in situations that predictably assure the theory's entailments
Inclusion Method	Restricting the theory by ruling in confirming cases

Table 3.2: Hayes-Roth's Methods

3.3.2 Representational Issues

In [Amarel, 1968], Amarel discusses how the representation of a problem can make an enormous difference to the difficulty of solving a goal in a theory, hence theory refinement can be achieved through changing representation. Amarel develops nine different changes of representation, each making the solution of the problem easier. These include steps such as translating the problem into more concise vocabulary, removing redundancies and searching for global patterns and symmetry. It is made clear that the representation of the problem has an enormous effect on its solvability, although it must be remembered that all of the changes come with a cost and it may be hard to predict in advance if these costs will be repaid. The changes are all equivalence preserving.

3.3.3 Symbolic Machine Learning and Inductive Programming

One approach to the problem of theory refinement is the use of machine learning. FORTE [Richards and Mooney, 1995] is a system which uses machine learning for revising function-free first-order Horn-clause knowledge bases. It is part of a growing body of work on inductive logic programming (ILP) [Muggleton, 1999]. In general, ILP systems that modify incorrect knowledge require interaction with the user [Shapiro, 1982], although FORTE is fully automated and finds a minimal revision of a theory to make it consistent with a set of training examples by performing hill search. EITHER [Ourston and Mooney, 1990] is a propositional theory refinement system which uses deduction, abduction and induction to refine a propositional Horn-clause theory. GOLEM [Muggleton and Feng, 1990] is based on Plotkin's idea of relative least general generalisations (RLGG), which Muggleton shows to be closely related to inverse resolution [Muggleton, 1999].

PAC (Probably Approximately Correct) learnability theories have been used by Mooney to approach the problem of theory revision [Mooney, 1995]. He uses the notion of syntactic distances, which describe the number of elementary operations needed to transform the given concept to the desired representation. He considers the PAC-learnability of the class of concepts having a bounded syntactic distance from a given concept representation, which allows for the concept of prior knowledge. Sloan and Turán expand on these ideas, formulating the problem in the model of learning with membership and equivalence queries [Sloan and Turán, 1999]. They present revision algorithms using these queries and show that they are efficient for certain types of formulae but are not efficient for other, more complex formulae. Koppel *et al* [Koppel et al., 1994] describe an approach, called PTR, which uses probabilities associated with domain theory elements to numerically track the 'flow' of proof through the theory. PTR is proved to converge to a theory which correctly classifies all examples.

Another approach to machine learning is explanation-based learning (EBL). In this ap-

proach, machines create justified generalisations from single training instances. Their ability to generalise from a single example follows from their ability to explain why the training example is a member of the concept being learnt [Mitchell et al., 1986]. In order to do this, they rely heavily on background information, as opposed to the empirical approach where minimal background knowledge is assumed [Ellman, 1989]. EBL can be seen as a method that performs four different learning tasks: generalisation, chunking, operationalisation, and analogy.

3.3.4 Correcting Faulty Formulae

Another relevant approach is work on correcting faulty formulae. A formula is considered to be faulty if it is a contingent formula that was expected or intended to be a theorem [Monroy and Bundy, 2001]. In his thesis, Moore introduced a methods that attempts to avoid over generalisation [Moore, 1974]. Formulae sometimes need to be generalised in order to be proved by induction. However, generalisations may not always be sound: the generalisation of a valid formula may not itself be valid, because the common subterms that have been generalised may be important. For example, *(Sort (Sort A)) = (Sort A)* is a valid formula which is generalised to *(Sort Genrl1) = Genrl1*, where *Genrl1* is a skolem constant which replaces the subterm *(Sort A)*. This generalised formula is only true in certain circumstances: specifically, if Genrl1 is an ordered list. Moore's system attempts to create conditions under which these formula, not true in the general case, can be made true. In this case, this would be *(Ordered Genrl1) → ((Sort Genrl1) = Genrl1)*. However, this approach requires both the original formula and the over generalised one, so as to determine what the restriction on the generalised case should be, and thus does not deal with the problem of correcting faulty formulae in general. This approach was further developed by Frańová and Kadratoff [Frańová and Kodratoff, 1992]. They developed a method, *PreS*, that builds a corrective condition using the constructive principle of proofs-as-programs. How-

ever, this can only handle certain types of conjectures. Protzen applied PreS to find conditional, recursive corrective formulae [Protzen, 1996].

Monroy stated the problem of correcting faulty formulae G as the problem of finding a corrective condition P such that $P \rightarrow G$ is a theorem [Monroy, 1999]. The mechanism used to find this condition P is based on abduction. Building a corrective condition amounts to providing an algorithm for computing the condition. The mechanism renders an algorithm as a collection of equational cases, each of which might be conditional. Abduction is used to explore the associated partial proof trees. These trees can be extremely large and sometimes infinite, and care must be taken to avoid combinatorial explosion. Proof planning is used to address this problem (see Section 2.6.1). The abductive mechanism is given as a set of heuristics. Each one captures the restricted way in which the search for a proof of a faulty formula can fail and provides knowledge to recover from such a failure.

3.3.5 Program Debugging

There has also been some work done on the problem of automated program debugging and repair. In his work on Cynthia [Whittle et al., 1997], Whittle approached the problem of debugging programs by considering the analogous problem of theory revision. Hence when changes were made to the top-level program, the effects of that change on the underlying theory were assessed and the theory repaired accordingly. Examples of these changes include renaming items, changing the type of items, for example from list to tree and changing patterns inherent in the type: for example, the base case for a list is `nil`, whereas for a tree it is `one`. This allows users to move from one program to another in a way that is verifiably valid. Users are only permitted to make a limited number of predefined changes, hence the way in which things will need to be changed is predictable. An earlier implementation of similar ideas can be found in the *recursion editor* [Bundy et al., 1991], which is designed to edit Prolog programs.

Shapiro also did much work in program debugging [Shapiro, 1982] and provided algorithms for use, although he did not consider how it might be analogous to theory revision. He divides the problem of program debugging into two different aspects:

- How do we identify a bug in a program that behaves incorrectly?
- How do we fix a bug once it is identified?

Shapiro's MIS system uses *diagnosis algorithms* to address the first issue and *bug-correction algorithms* to address the second. Unlike many of the theory revision systems, the MIS system [Shapiro, 1982] is capable of specialising theories as well as generalising. MIS requires user interaction, as the user is queried for information about the intended behaviour of the program.

Shapiro developed three different kinds of diagnosis algorithms, one for termination with incorrect output, one for termination with no output and one for non-termination. We reproduce here his diagnostic algorithm for termination with incorrect output, as this is the most relevant for our work. The idea of the algorithm is to step through the procedural calls of the computation to discover which is the first to return an incorrect output, thus identifying it as incorrect. The algorithm uses a *ground oracle* for an interpretation **M**, such that, on input $< \mathbf{p}, x, y >$, it outputs *yes* if $< \mathbf{p}, x, y >$ is in **M**, and *no* otherwise, where **p** is a procedure and x and y are vectors over some domain **D**:

Algorithm:

Input: A procedure **p** in **P** and an input x such that **p** on x returns an output $y \neq \perp$ incorrect in **M**.

Output: A triple $< \mathbf{q}, u, v >$ not in **M** such that **q** covers $< \mathbf{q}, u, v >$.

Algorithm: Simulate the execution of **p** on x that returns y; whenever a procedure call $< \mathbf{q}, u >$ returns an output v, check, using a ground oracle, whether $< \mathbf{q}, u, v >$ is in **M**. If it is not, return $< \mathbf{q}, u, v >$ and terminate.

3.4 Fault-Catalogue and Model-Based Diagnosis

Another field which addresses the problem of mismatches and how to deal with them is the field of diagnosis. It is concerned with the problem of inconsistent information in databases, terminologies or ontologies, and with how these inconsistencies can be identified and resolved. Like our approach, this is concerned with diagnosing what the problem is by considering what the potential problems could be and, in some cases, considering what could be done to rectify the problem. There are two main approaches within this field: fault-catalogue based diagnosis and model-based diagnosis. The first relies on a list of potential faults to aid in diagnosing the problem, whereas the second relies on a model of the system and attempts to compare actual behaviour with the model to determine how it may explain inconsistencies.

A seminal paper in the field is [Reiter, 1987], which presents a general algorithm for computing all diagnosis for a given faulty system. This is developed for a first-order representation language, but the theory is reasonably general and is compatible with many other logics. The representational logic specifies how the system would normally behave if all its components are working properly. The system failure can then be explained in terms of some component working abnormally, by calculating what kind of abnormal component behaviour would lead to the observed behaviour of the system. The algorithm computes all possible diagnoses for a given faulty system, and it is then left to the user to determine which diagnosis was actually responsible for the fault out of the candidate combinations; this is usually done experimentally. A system is considered to be a pair: $system = (Sd, Cmp)$, where Sd is a system description and Cmp is the set of system components. [de Kleer and Williams, 1987] presents a general, domain-independent diagnostic engine for dealing with multiple faults by using a model to describe the physical structure of a device in terms of its constituents; thus the diagnosis problem becomes that of determining how the device differs from its model. The diagnostic process requires a notion of the physical structure of the device,

models for each constituent, a set of possible model-artifact differences and a set of measurements to determine how each constituent is performing, and it produces a set of candidates to explain the discrepancy between the behaviour predicted by the model and the experienced behaviour of the device. The method of computing diagnoses is based on notions of *conflict sets*. A conflict set describes all the possible combinations of components that could have lead to the fault. These may be single fault diagnoses (failure of a single component would explain the fault) or multiple fault diagnoses (failure of more than one component in combination is necessary to explain the fault). Single fault diagnoses are often preferred, as they provide a simpler explanation - *i.e.*, it is more likely that a single component has failed to cause the fault than that the specific combinations of multiple components have simultaneously failed. Conflict recognition is performed to identify all the minimal conflict sets, and a set of minimal candidates is generated from these minimal conflict sets through experimentation.

The philosophy of model-based diagnosis [Console and Dressler, 1999] is that diagnosis should be based on an objective model of the device (system) to be diagnosed. Different types of models can be considered: for example, structural (concerning physical or logical structure); functional (describing function); behavioural (describing how functions are achieved); and teleological (describing the purposes of the use of the device); or a combination of approaches can be used. Clearly, the actual diagnostic process depends on the models, which are usually component-oriented. Devices are described in terms of minimal replaceable or repairable components. Component models include a list of variables and a definition of its mode of behaviour. The behaviour of components is described via a set of relations or constraints. Both the correct behaviour and a set of known faulty behaviours can be described. Different notions of explanation can be adopted, from weak ones based on consistency to stronger ones based on abduction.

More recently, these ideas have been developed by Scholbach and others. [Scholbach

and Cornet, 2003] describes diagnosis for conflicting ALC T-Boxes. The work is motivated by the example of DICE terminology (Diagnosis for Intensive Care Evaluation); DICE defines more than 2400 concepts and uses 45 relations, so the potential for error is large. Debugging conflicts requires both an *explanation* of the logical incorrectness and its *correction*. This paper focuses on the former, by attempting to find a single axiom in the TBox that is causing the contradiction. Minimal subsets of axioms of an incoherent terminology that preserve unsatisfiability of a particular concept are found by removing axioms that are irrelevant to the incoherence (that is, the TBox remains inconsistent after they are removed). This allows the exact position of the contradictions within the remaining axioms to be found. [Scholbach, 2005] describes a framework for debugging logically contradicting terms which is based on model-based diagnosis. Three different types of diagnosis, based on Reiter's hitting set algorithm [Reiter, 1987], which discover different types of conflict sets. The first is a DL-based reasoner (RACER), which is used to return entire terminologies as maximal conflict sets. However, this fails even on small incoherent terminologies. The second idea uses internal information from unsatisfiability proofs to return small (but not minimal) conflict sets. The third approach implements the specialised algorithms described in [Scholbach, 2005] to calculate minimal conflict sets. Evaluation indicates that the complexity of the general problem is so high that implicit information on proofs is necessary to make the problem tractable. The two more specialised algorithms for finding small and minimal sets work in practice, implying that implicit information on proofs is required.

Related work on debugging inconsistent ontologies and detangling taxonomies through diagnosing inconsistencies has been done as part of the MINDSWAP project [Cuenca Grau et al., 2004, Parsia et al., 2005]. A distinction is drawn between glass box approaches, where information from internals is extracted and presented to the user, and black box approaches, where the user acts as an oracle. When clashing information is discovered, attempts are made to find a set of axioms which support this clashing information, and hence which must contain an inconsistency. This is then

presented to the user. Attempts to fix the problem are focused on making it easier to for the modeller to understand what the problem is; thus, this is done interactively rather than automatically. [Rector et al., 2001, Rector, 2002] describe work on tangled taxonomies that focuses on maintaining taxonomies so that they do not become tangled in the first place.

3.5 Relevance to the Project

- **What the project requires from the field**

  A thorough grounding in the state-of-the-art technology in this field is a prerequisite for the project, as this is the context within which our work is based. It is necessary for understanding which problems remain unsolved, and for understanding how our work is related to the work of others. However, although we have taken inspiration from many of the approaches described above, we do not consider our work to be based on or a direct extension of any of them, but rather it is a new approach to the problem.

- **What the project contributes to the field**

  We believe that our work constitutes a novel approach to the problem which provides solutions and potential solutions for aspects of mismatch and refinement that have not otherwise been solved, and which complements the contributions of much of the work discussed in this chapter. A more detailed discussion of how our approach provides this, and how it relates to the approaches discussed in this chapter, is not possible before a more thorough explanation of our work has been provided. Such a discussion can be found in Section 7.5.

Chapter 4

Overview of the Ontology Refinement System

4.1 Overview

In this chapter we give an overview of the project and of the Ontology Refinement System (ORS): what we are attempting, how we intend to achieve this and why these approaches were chosen. We describe the aims and limitations of the project and briefly discuss how the limitations of the system can be minimised; this is discussed in further detail in Chapter 8. We introduce the subsystems of the overall system, discussing their role and how they interact with each other. Worked examples are given throughout to illustrate how the system works.

4.1.1 Research Goals

The overall aim of the project is to create an agent communication system in which, if interaction breaks down due to communication failure, agents are capable of detecting

the ontological mismatches that caused this problem and refining their ontologies to remove these mismatches in an appropriate manner. Communication failure occurs when agent interaction does not progress as the agents expect it to, and when this failure is due to ontological differences between the agents and not due to network, software or hardware failure. Once the problem has been identified and fixed, communication is then resumed using the updated ontologies. From this point on, this particular communication problem should not be encountered, although perhaps further communication problems will occur at a different point. This is envisaged as a dynamic, incremental system, where each communication problem leads to immediate diagnosis and refinement. The ramifications of this particular problem with respect to the rest of the ontology are considered: altering one part of the ontology will often entail changes in other parts of the ontology. Diagnosis and refinement of ontological mismatches is always prompted by failure: we do not compare the ontologies of the interacting agents except where parts of them are revealed through communication, nor do we examine unexpected locutions except where these are implicated in communication failure.

The end result is a system in which agents are continually altering their ontologies so that they can successfully interact with other agents, even if these agents do not have identical ontologies to them. If refinement succeeds, the output of the system is an indication that the desired objective of the communication has been fulfilled, together with an ontology that may be slightly different from the initial ontology, according to the refinement requirements of any problems encountered during the communication process, and a history of which refinements have been performed. Sometimes it is not possible to perform the appropriate refinement. In such a case, the output is a diagnosis of the problem, a description of why it was impossible to refine the problem, an ontology that may differ from the original ontology if previous, successful refinements were performed, and a history of which refinements have already been performed on the system before further refinement became impossible.

4.1.2 Scope of the Project

The scope of the project includes many different research areas, each with complex and difficult issues of their own. For example, the complexities surrounding automated communication between agents are immense. The problems of ontological mappings, mergings and alignment are currently amongst the most difficult issues faced in the agent community, and the push to develop the Semantic Web has led to these becoming important areas of research. Thus the project needs to find a balance between an attempt to understand these issues and deal with them as far as possible on the one hand, and a central focus on the themes of diagnosis and refinement, without too much diversion into sub-issues, on the other.

A key part of this project is therefore to find parameters within which to investigate this problem such that it is possible to produce interesting and informative output. It is important in such a large domain to carefully control the direction of the research to ensure that it remains focused, and to this end we have identified a particular approach to the problem, and a particular subset of issues that we wish to address. We have made several important assumptions about agent behaviour which are discussed in Section 6.2. Some of these are bold assumptions and are the subject of major research effort within the agent community. However, by making these assumptions we are able to develop a platform from which to explore the issues that this thesis is really concerned with: the problem of ontological mismatch. Without making such assumptions, we would be unendingly caught up in tangential issues.

Among the key assumptions we have made is that the ontologies of agents will be largely similar, that agents behave in simple and understandable ways, that the agent that is attempting to refine its ontology will trust the ontologies of other agents, and that the agents are only involved with interactions with each other; that is, there are no agent interactions of which we are unaware. These assumptions create a tractable problem. Discussions about how this work can be expanded to remove some of these

assumptions is discussed in Chapter 8.

We therefore do not suggest that this research is a full solution to the problems that we are addressing, but rather that it is a solution to an interesting subset of these problems. We claim that it introduces techniques that can be shown to facilitate agent communication that would otherwise be impossible, and that these techniques have the potential to be developed further to create a more complete solution to the problem.

4.1.3 Project Domain

The context in which we explore the problem of ontological refinement is a planning environment. The agent communication system contains *planning agents* (PAs), which form plans and attempt to execute them through communication with *service-providing agents*. Thus a plan step is executed by persuading an appropriate agent to perform that step. Any agent in the system has the ability to be a planning agent or a service-providing agent, but we assume that only one plan is being executed at any one time, and thus only one agent is acting as a PA at any one time. Considering ontology refinement within a planning domain allows us to generate a situation in which the ontology mismatches are highlighted (through plan failure) and in which the positive effects of the refinement process can be verified (by showing that planning with the refined ontology can eventually result in the goal being achieved).

We are exploring the situation in which a PA is attempting to achieve goals by forming and executing plans in a given domain. The inputs and outputs of this agent are shown in Figure 4.1. The PA will form plans to achieve the goals which it is given, based on its understanding of the world. Each plan step will be executed through interaction with other agents. The PA initially makes the assumption that these other agents have the same ontology as it, and thus the conditions under which the PA believes an action is executable will match the conditions under which other agents will perform it.

Additionally, this assumption entails the assumption that the effects of a given action being performed will match the PA's expectations of the effects. It is very useful to make such assumptions, as this allows one to form plans based on one's understanding of the world. However, it is often the case that other agents may have ontologies that are similar but slightly out of sync. This happens, for example, because ontologies are dynamic and are regularly updated, leading to agents having different versions of the same ontology, and also because off-the-shelf ontologies are adapted by users to their particular needs. This can lead to plan execution failure, since the other agents may not always behave in the way predicted by the PA's expectations.

The system is fully automated. The PA will prompt initially for a goal, which is provided by the user, but after that there is no user interaction. If plan execution failure is experienced, the PA will diagnose and refine the problem and then replan. In some cases it is not possible to fully diagnose the causes of failure and thus to appropriately refine the ontology. In such situations, the attempt to reach the goal must be abandoned. Further discussion of this issue can be found in Section 5.4.

The decision to explore the problem of ontology refinement within such a context limits the applicability of the system to a planning environment. The system cannot function without this because the methods of identifying that mismatch has occurred, and diagnosing what the mismatch may be, depend on the planning context. However, the theory describing the kinds of ontological mismatches is not dependent on a planning context, and this theory could be reimplemented in a system that operated in a different context.

4.1.4 The Ontology and the Meta-Ontology

Each agent operating in a domain has both an ontology and a meta-ontology describing that domain. The ontology contains the ground-level information about the domain, and the meta-ontology contains information about those ground-level terms. In order

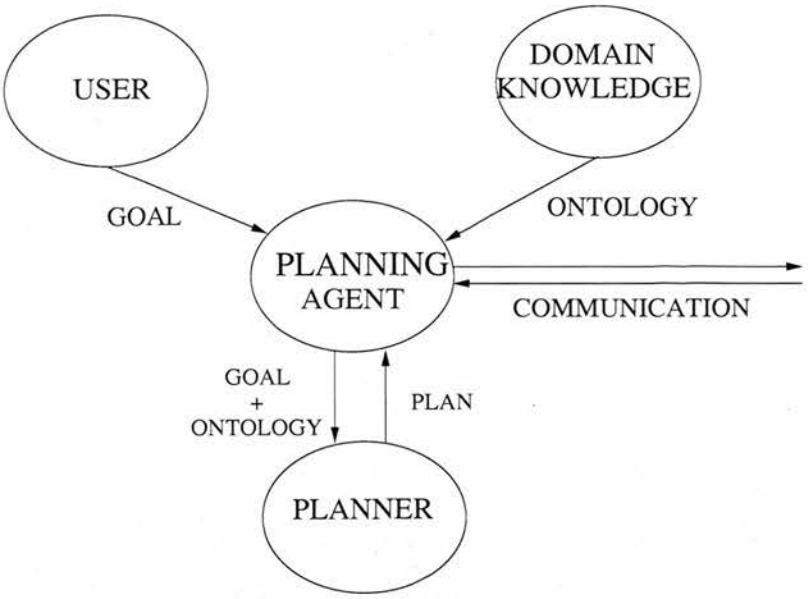

Figure 4.1: Inputs and Outputs of the PA

to be able to diagnose and perform the kinds of refinements we discuss in later sections in this chapter, we require information about the ontological objects within the ontology. For example, we require a class hierarchy not only of individuals, but also of predicates themselves: the predicate *(Dollars ?Amount)* may be a subclass of the predicate *(Money ?Amount)*. We require information about the predicates concerning how they ought to be interpreted in plan execution: for example, facts that need to be checked with other agents, facts for which the PA depends on its own information, and so on. We also require information about the rules: for example, which agent is able to perform each action. This is included in the meta-ontology.

The ontology and the meta-ontology have both signature and theory objects, with some of the signature objects in the ontology being represented as theory objects in the meta-ontology.

Naturally, we cannot expect every agent that is encountered to have a meta-ontology, as this is not standard. Many ontology languages allow such meta information to be contained in the ground-level ontology. This does not mean that the system will not work with such agents, only that they will not be able to diagnose certain types of refine-

ments, because they will not have the necessary information about how the predicates are related. In many ontology languages, this kind of relationship can be described in an object level ontology: for example, by using the *subPropertyOf* relation in OWL. Thus a meta-level ontology would not be necessary for describing the predicate hierarchy.

4.2 Worked Example

We introduce an example which is intended to illustrate as many as possible of the diagnostic and refinement techniques described in this chapter[1]. It is also intended to be typical of the kind of scenario that one might expect to encounter on the Semantic Web. It is based on the Semantic Web Conference ontology, parts of which can be found in full in Appendix A, and which can be found in full on the project website[2]. This ontology concerns a virtual agent whose role is to form plans to achieve given goals, and execute these plans in a Semantic Web like environment, where other agents are available to perform tasks or services. In particular, this PA is able to organise trips to conferences.

In this section, we introduce the general background of the example, and then give precise examples of refinements in the appropriate sections.

Consider the situation in which a researcher wishes to present a paper at a conference. Since he has a lot of demands on his time, he does not wish to spend time organising his visit to this conference, and instead passes an electronic copy of his paper to an autonomous agent, together with information about which conference he wishes to attend, and allows this agent to perform the necessary administration for him. Once the agent has the goal, it will attempt to form a plan to achieve this goal. In this situation,

[1] Output from the ORS can be found in Chapter 7

[2] `http://dream.inf.ed.ac.uk/projects/dor/`

this plan will involve, at a simple level, converting an accepted paper so that it is in the format required for final submission, submitting the paper to the conference, registering for the conference, booking accommodation, booking flights from the researcher's home city to the city where the conference is located, and perhaps submitting a request for reimbursement from lab funds once the costs have been paid. A more detailed plan may involve arranging transport from the researcher's home to the airport, arranging transport the other end, registering for workshops, and so on. For the purposes of clarity, we have kept the example concise.

In this example, the agent receives a goal:
(PresentedAtConference Researcher Ai-Conf My-Paper) and a paper, *My-Paper.ps*. The goal predicate takes three arguments: a conference attendee name, *Researcher*; a conference name, *Ai-Conf*; and a paper name, *My-Paper*. The predicate definition contained in the ontology specifies that the class of the third argument must be *Paper*. However, it does not specify the subclass of the class *Paper*, e.g., *PsPaper*, *PdfPaper*: this is determined by the submission process and may vary from conference to conference. The goal is simply to submit that paper in the required form; the conversion into the required form is handled by the agent as part of the plan.

The following plan is produced:

[(ConvertPaper Researcher My-Paper.ps My-Paper.pdf),
(SubmitPaper Researcher My-Paper.pdf Ai-Conf),
(Register Researcher Ai-Conf Registration-Cost),
(BookAccom Researcher Ai-Conf Accommodation-Cost),
(BookFlight Researcher Ai-Conf Flight-Cost),
(Reimburse Researcher Ai-Conf Registration-Cost Accomodation-Cost Flight-Cost)].

Once the plan has been generated, it is executed step-by-step within the agent communication system. The PA will first locate the appropriate agent with which to converse, for example a *Paper-Conversion agent* may be required for the action *(ConvertPaper*

Researcher My-Paper.ps My-Paper.pdf). The PA will request that the appropriate agent performs the action, and then wait for a response. This response will either be an indication of whether the task has succeeded or failed, or it will be a request for further information. An agent will carry out a task that it can perform if it is able to under the circumstances; that is, if the preconditions for that task are fulfilled. The agent will be able to verify some of the preconditions for itself, will have to check with other agents to verify some of them, and the validity of other preconditions will only be determinable though questioning of the PA.

For example, when attempting to register with the *Registration agent*, the conversation may proceed as follows (note that PA represents the planning agent, and RA, the registration agent; arguments beginning with a question mark are variables, and otherwise are constants):

PA: **request** (*Register Researcher Ai-Conf*)

RA checks those of its own preconditions that it can, plus any that must be verified with the other agent, and then verifies the others with PA

RA: **query** (*Accepted-Paper Researcher Ai-Conf ?RefNo*)

PA: **reply** (*Accepted-Paper Researcher Ai-Conf 125*)

RA verifies that this is an acceptable reference number

RA: **performed** (*Register Researcher Ai-Conf*)

Registration information is now generated for the researcher; PA needs to know this information to continue.

PA: **query** (*Registered Ai-Conf Researcher ?RegistrationNo*)

RA: **solution** (*Registered Ai-Conf Researcher 539*)

There are two stages to the above communication. The request itself does not contain any reference to the registration number that is given to the PA at the end of the interaction. No reference to this is necessary in the action because it is not required in order to perform the action, it is merely an effect of the action. In order to determine

whether the action can be performed for this agent, the RA must verify that the PA is requesting this information on behalf of someone who already has a paper accepted. The PA must then check its effects are fully instantiated before updating its ontology. In order to instantiate its effects for this action, the PA then directly asks the RA to return the relevant registration reference number. Since the action has been performed successfully, this information is available; otherwise it would not be.

Unless unexpected problems arise, this process will continue until each step of the plan has been successfully executed through communication with appropriate agents. The successful execution of the final step of the plan entails the goal being fulfilled.

However, the belief that the generated plan is a viable list of steps to achieve the goal is based on the assumption that the ontology from which the plan was formed is an accurate representation of the domain in which the plan will be executed. If the ontology of the planning agent is in fact incomplete or incorrect with respect to the ontologies of the other agents with which it is interacting, unexpected problems may occur during the execution of the plan. For example, in the above conversation, consider the case in which the *Accepted-Paper* predicate had been altered to contain information not only about the attendee and the conference, but also about the paper which they were presenting:

(Accepted-Paper ?Attendee ?Conference ?Paper ?RegNo).

However, the PA is using an out of date version of the ontology, and thus considers *Accepted-Paper* to be a ternary predicate. In response to the question from the above dialogue:

AA: **query** (*(Accepted-Paper Researcher Ai-Conf ?Paper ?RegNo)*)

the PA will reply *No*, since it does not have any facts that match, and the plan step will fail to be executed.

Throughout the rest of the chapter, we describe how our system will deal with such a failure; how the ontological mismatch that caused this will be identified, and how the

PA's ontology will be refined so that this mismatch no longer exists. Examples of the types of refinements we might expect to encounter, together with descriptions of how they might be refined, are given in the appropriate sections below.

This example contains many assumptions about agent behaviour; these are discussed in more detail in Section 6.2.

4.3 Sub-systems of the Ontology Refinement System

The ontology refinement system (ORS) consists of various different sub-systems: the *refinement system*, the *diagnostic system*, the *agent communication system*, the *planning system* and the *translation system*. The PA exists as part of the agent communication system and is able to call the other parts of the system as necessary. Figure 4.2 shows the interaction between the subsystems.

The PA will firstly wait for a goal, and as soon as one is received, it will commence attempting to achieve it. The ontology is translated into a suitable format so that it can be interpreted by the planner and directly interpreted by the PA and the plan deconstructor. The planning version of the ontology is then sent to the planner, and a plan is returned. Sometimes the planner will fail to return a plan. It might be that it is not possible to achieve the goal using the PA's original ontology, or it might be, if refinements have already been made to the ontology, that these refinements have made it impossible to form a plan. If the planner does not return a plan, the process ends in failure: it is not possible to achieve the goal. If a plan is produced, this plan is deconstructed to produce a justified plan, which links the plan to the underlying ontology (this process is discussed in Section 6.3), and the PA then attempts to execute the plan. If this is successful, the PA will then ensure that the ontology is updated with respect to the results of the plan, and the process will terminate successfully. If failure is encountered, the agent attempts to diagnose the underlying ontological mismatch that caused the

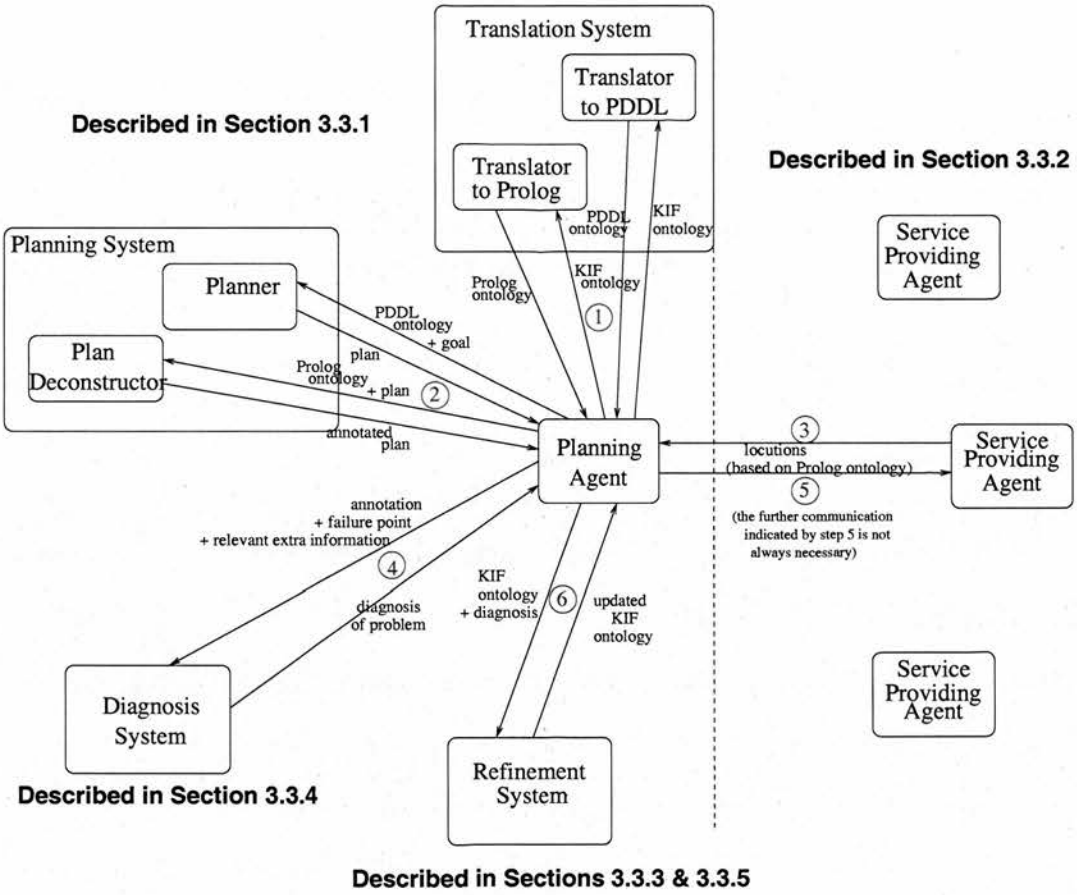

Figure 4.2: Architecture and Interaction of the ORS

failure. In some cases, it is not possible to fully diagnose the mismatch; in such cases, all the PA can do is to mark the relevant part of the ontology (for example, the relevant action rule) as unusable. If diagnosis is successful, the ontology is refined accordingly. The updated ontology is then retranslated for the planner, and the process is repeated.

These last two processes, diagnosis and refinement, are the central concern of the project. The rest of the ORS provides the structure within which diagnosis and refinement can occur.

In summary, there are five different modules in the system:

1. The diagnostic system
2. The refinement system
3. The agent communication system
4. The planning system, which consists of:
   (a) The planner
   (b) The plan deconstructor
   (c) The plan finder, which extracts and translates a usable plan from the planner output
5. The translation system, which consists of:
   (a) The translator from KIF to PDDL
   (b) The translator from KIF to Prolog
   (c) Various smaller translation processes

Of these components, the planner and the agent platform are taken off the shelf (this is discussed further in the relevant sections); all the other components have been written for the project. All of these bespoke components are written in Sicstus Prolog [Sicstus, 2005].

4.4 Flow of System

The system is controlled by the PA. The flow of control is illustrated in Figure 4.3. The PA will first of all prompt for a goal, and then translate the ontology and the goal into the PDDL representation and the Prolog representation (discussed in Section 4.5). The goal is required during translation so that it can be placed into the PDDL representation and used during planning; it is not used again unless plan execution fails and, after diagnosis and refinement, a new plan has to be formed.

Once the translation process has created the PDDL and the Prolog versions of the ontology, the PA then calls the planner using the PDDL representation. If the planner fails to return a plan, this is reported to the PA, and the process fails. This will occur if it is not possible to reach the goal from the initial state described in the ontology, using the actions described in the ontology. This may happen the first time that the agent attempts to form a plan, or it may be that it was initially possible to form a plan, but this plan failed to be successfully executed, and refinements made to the ontology, as a result of that failure, resulted in a situation where it was no longer possible to reach the goal. This process is illustrated in Figure 4.4.

If the planner succeeds in returning a plan, this plan is first interpreted and translated so that it can be read by the PA. The PA then attempts to execute the plan step by step. At each step, it locates the appropriate service-providing agent to perform the task, sends a request to that agent to perform the task and waits for one of the following responses:

1. Information about the final outcome:
    - An indication that the action has been successfully performed,;
    - An indication that the action has failed to be performed;
2. A request for further information:

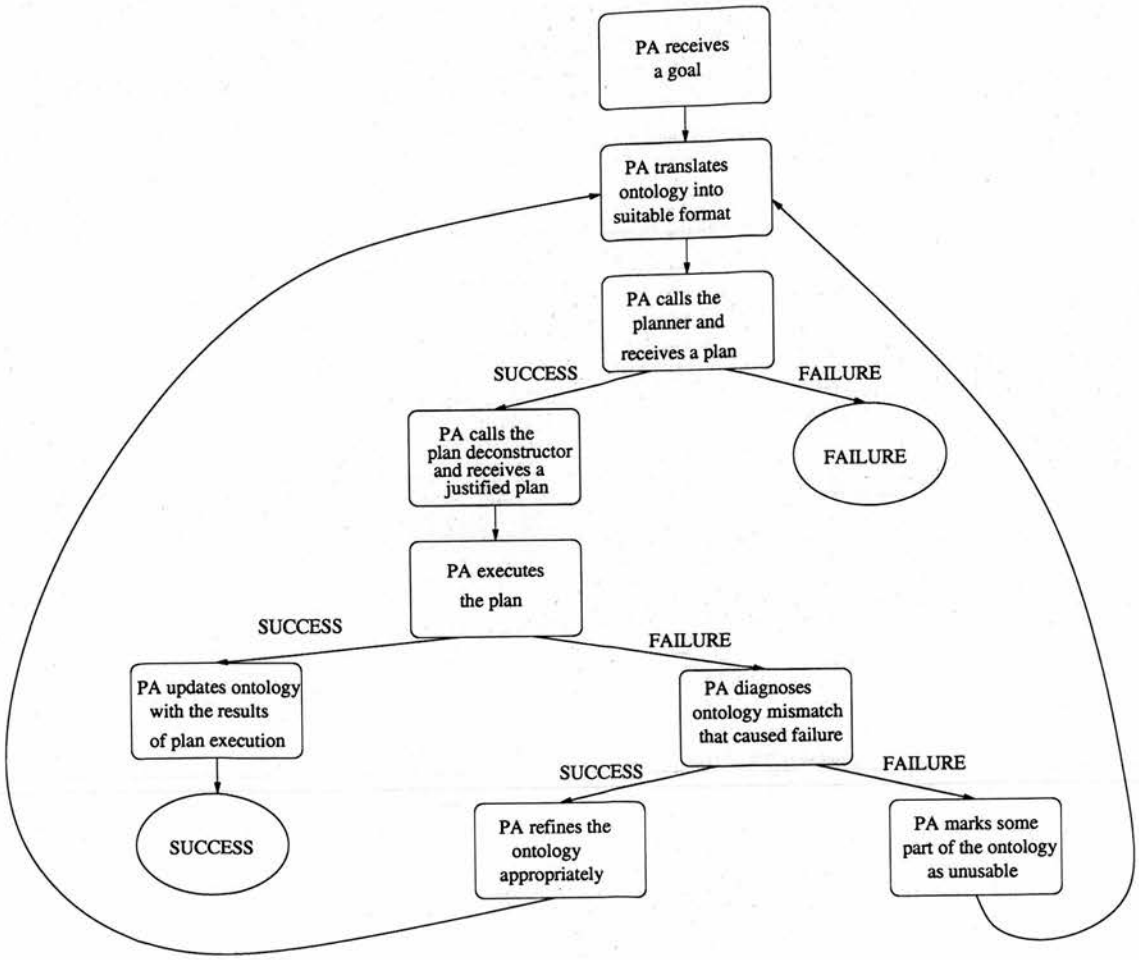

Figure 4.3: Flow of Control of the ORS

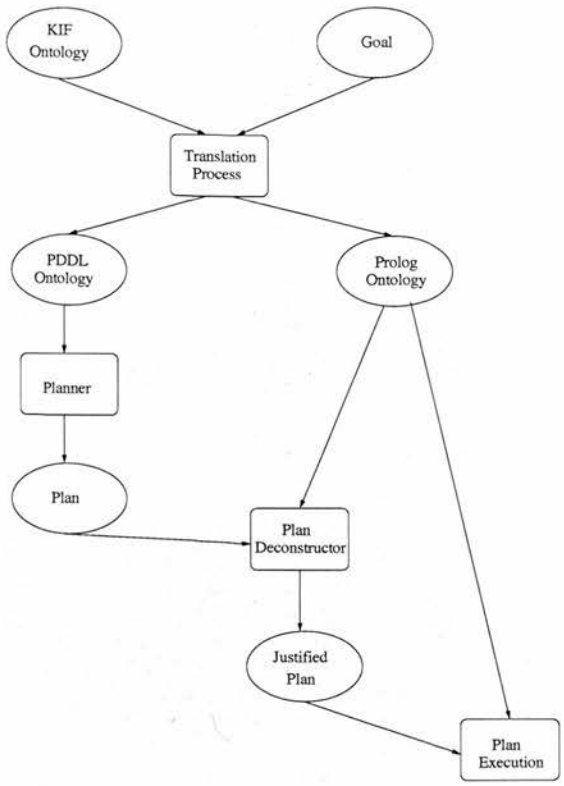

Figure 4.4: Flow of System

- A fully instantiated predicate, for which the PA must give the truth value according to its ontology; an agent will either reply that this is true if it has information to indicate this, or that it cannot confirm the truth if it does not have such information;
- A fully or partially uninstantiated predicate, which the PA must instantiate according to its ontology. If there is no fact in its ontology that corresponds with this, it replies that this is false.

If the response received from the service-providing agent is of type 2, the PA, after responding appropriately to it, waits again for one of the above responses. This loop continues until a response of type 1 is received. If the action succeeds, the PA then attempts the next step of the plan, until the goal is reached. At that point, the PA returns output to indicate that the plan has succeeded, and terminates. If all the actions

succeed, the system is just performing planning within a multi-agent system. It is when failure occurs that the core abilities of the ORS are called into use: diagnosis and refinement. If any action fails, the PA calls the diagnostic system to receive a diagnosis of the problem. If a precise diagnosis is possible, the refinement system is then called to implement the diagnosis (discussed in Section 5.5). In cases where no refinement can be performed, the process terminates, with the PA providing feedback that the goal could not be achieved and detailing why a precise diagnosis could not be performed. If an appropriate refinement can be performed, which may be either the alteration of an ontological object or an indication that this object should not be used in planning, the process is then repeated. The updated ontology is retranslated, and a new plan is formed, if possible: the refining of the ontology could mean that it is impossible to form a plan to achieve the goal. If a plan is produced, this is then executed step by step in the manner described above. The process is repeated until the goal is achieved, or until diagnosis, refinement or replanning proves impossible.

4.5 Representation

The way in which the ontology is represented is one of the key features of the system. There are three main things that are required of the representation:

1. To be a recognised format for agent ontologies, so as to make the application as broad as possible,

2. To be usable by an efficient planner so that the PA can use its ontology to find solutions to achieving goals,

3. To be available for constant reference and updating within an agent communication system.

It is not possible to satisfy all three of these requirements using the same representation. There is no common agent ontology language that is also the representation required by a state-of-the-art planner. Additionally, we wish to keep the agent communication system simple, and to that end, it is written in Prolog and based on the Linda server (see Section 6.2). Therefore, these three aims are achieved by using three different representations:

1. The original ontology is written in KIF, developed on the Ontolingua server;
2. The planning representation is written in PDDL;
3. The ontologies with which the agents work within the agent communication system are written in Prolog. This is also used by the plan deconstructor.

It should be noted that we use a restricted form of KIF in order to constrain the problem. The KIF ontologies ORS can deal with are restricted in the following ways:

- **Classes:** As KIF is an expressive ontological language, it allows for complex class definitions. However, our restricted KIF ontologies are permitted only simple class definitions that name the class and place it in the class hierarchy. Specifically, full KIF allows the following aspects of class definitions that are not allowed by our restricted KIF:
  - Cardinality restrictions;
  - Statements about disjointedness of classes;
  - Declaration of special characteristics of classes such as inverse, uniqueness, etc.;

  Note that some of these properties may be declared in the meta-ontology, where it is possible to talk about classes as objects. However, this information cannot be contained within the class definition, as is possible in full KIF.

- **Quantification:** In our restricted KIF, all variables are implicitly universally quantified over finite domains.

These restrictions certainly limit the applicability of the system, since they limit the expressiveness of the ontologies. It has already been established through the need to build OWL on top of RDF-S that such expressivity is necessary in useful agents [Antoniou and van Harmelen, 2004]. In fact, some of the ways in which the expressivity of OWL improves the expressivity of RDF-S are similar to the class expressivity differences between full KIF and our restricted KIF. Thus removing these assumptions is an important part of further work (see Section 8.1.1). However, we believe that using these restrictions for ORS was justified, because it was necessary for us to constrain the problem so that a working solution could be developed, and this seemed an obvious restriction to use due to the planning context. Most planning problems and representations, and most modern planners use a similar simple approach to class definitions, where they are merely used to define a class hierarchy and restrict the kind of object that can be used in certain plan steps. Thus our approach allows a fairly full investigation of our chosen domain, but we have not included the extra expressivity that would be necessary for other forms of agent interaction.

It could be argued that, since the expressivity constraints on our restricted KIF and on RDF Schema are in some ways similar, we would have been better to use ontologies written in RDF Schema. The obvious advantage of this is that this is a much more common representation language for the Semantic Web, and if we improve the expressivity of the representation, as we envisage doing to convert our restricted KIF into full KIF, this would result in OWL ontologies, which are generally considered to be the standard representation for Semantic Web agents [DAML, 2002]. The main reason we choose not to do this is that KIF, even in our restricted version, does have some expressive advantages over RDF Schema and OWL. In particular, it allows predicates of unlimited arity, rather than the triples (binary predicates) that RDF Schema

and OWL are restricted to, and the use of variables allows for the definition of general axioms, which can be used to describe actions (see below for a discussion of how this is done). Both of these aspects are important in the planning domain. Additionally, the semantics of RDFS are complex and based on graph theory rather than Description Logics. If ORS were to be used in a more general Semantic Web like setting, without the restriction to the planning domain, then utilising the accepted ontology representation language (currently OWL) would become more important than taking advantage of the increased expressivity of (even restricted) KIF. A discussion of what would be involved in changing ORS to work with OWL ontologies can be found in Section 8.1.2.

Ontolingua is an ontology server designed to facilitate reuse and sharing of ontologies [Farquhar et al., 1996, Gruber, 1992]. It consists of a set of tools and services to support the process of achieving consensus on shared ontologies by geographically distributed groups. The ontology editor uses the World Wide Web to enable wide access and provide users with the ability to publish, browse, create and edit ontologies. We chose to do this because this ensures that the ontologies are developed and expressed in a specific manner, thus easing the issues of translation and refinement. It also facilitates the process of creating a KIF ontology, and ensures that any ontology produced is consistent. The Ontolingua server builds KIF ontologies on top of the frame ontology [Frame Ontology, 1999]. The frame ontology defines the terms that capture conventions used in object-centred knowledge representation systems. Since these terms are built on the semantics of KIF, one can think of KIF plus the frame-ontology as a specialised representation language.

The KIF ontology is viewed as the original ontology, and translations to the other representations are performed as necessary. In the system, this need is initially prompted by the existence of a goal. When a goal is passed to the system, the system reacts to this by translating the KIF ontology into both PDDL and Prolog. The PDDL ontology is used to produce a plan and is then discarded; the Prolog ontology is used by the

agent during plan deconstruction and plan execution, and can then also be discarded. If refinements are necessary, these are performed directly on the KIF ontology. The PDDL and Prolog ontologies are not altered, since these have been discarded, and if a further goal is received, or the original goal is not yet reached, these will be regenerated from the new KIF ontology. Some maintenance of the ontology is also necessary during ordinary plan execution (*i.e.*, even excluding refinement situations). This concerns facts that are made true or made false as plan steps are executed. For example, after a *Buy* action, *(Money PA 100)*, which was previously true, will become false, and *(Has PA Item)* and *(Money PA NewAmount)*, where the value of *NewAmount* will be determined by the cost of the item, will become true. During plan execution, these updates are made only on the Prolog ontology (which is the working ontology at this stage), and a record of all these changes kept. There is no communication with the KIF ontology during plan execution. When plan execution terminates, either because the goal has been achieved or because plan execution failure has been encountered, the KIF ontology is updated accordingly.

As discussed in Section 2.3.1, many ontology languages are not very suitable for describing services. However, in KIF it is possible to define axioms, which are essentially inference rules describing a conjunction of conditions that entail another conjunction of conditions. For example:

(Parent ?X ?Y) $\wedge$ *(Female ?X)* $\rightarrow$ *(Mother ?X ?Y).*

We have used these axioms to describe action rules. However, there is a potential problem with doing this, because an action rule implies a time ordering: first, the conjunction on the left of the rule is true, then the action occurs, then the conjunction on the right of the rule is true. In an inference rule there is no time element, and the conjunction on the left and right of the rule, if true, are both true simultaneously.

In order to use inference rules as action rules without introducing logical inconsistency, we have therefore used Situation Calculus [McCarthy and Hayes, 1969]. This intro-

duces a *situational argument* to *fluents* (predicates whose value changes depending on the situation). Using Situation Calculus, a *Buy* rule might be stated as follows

(Money ?Agent ?Amount ?Situation1) $\wedge$ *(Cost ?Item ?Price)* $\rightarrow$
(Has ?Agent ?Item [Buy, ?Situation1]) $\wedge$ *(= ?NewAmount (- ?Amount ?Price))*
$\wedge$ *(Money ?Agent ?NewAmount [Buy, ?Situation1])*

Thus *(Money ?Agent ?Amount ?Situation1)* and *(Money ?Agent ?NewAmount [Buy, ?Situation1])* are distinguished by the situational argument, and even if *?Amount* and *?NewAmount* have different values, the rule is not inconsistent, as would be the case if the situational arguments were not present.

A situation is a stack of the actions that have been performed when the fact is true. Thus if no previous actions had been performed, *Situation1* would be $[Start]$ and the next situation, *[Buy, ?Situation1]*, would be $[Buy, Start]$.

4.5.1 The Meta-Level Ontology

We have introduced various meta-level predicates that contain information about the predicates in the ontology. We consider the ontology to consist of two parts: an object-level ontology, which contains all the information pertinent to the particular situation, and a meta-level ontology, which contains information about the objects that may appear in the object-level ontology. The meta-level ontology contains the same kinds of objects that are found in the object-level ontology: relations, functions, classes, a class hierarchy, individuals and actions. The way an object is classified differs from the object-level to the meta-level ontology. Relations, functions and actions in the object-level ontology are considered to be individuals in the meta-level ontology. This means that we can make statements about them that are not otherwise possible; for example, we can give them classes. This enables us to define a predicate hierarchy, and to give relations specific classes like the ones mentioned below. It also means that relations,

functions or actions can be mentioned within facts. For example, we can create the fact *(Agent-Needed Ticket-Selling-Agent Buy-Ticket)* to indicate that the agent needed to perform the action *Buy-Ticket* is *Ticket-Selling-Agent*. We can only state this in the meta-level ontology, where actions are considered to be individuals of class *action*. In the object-level ontology this cannot be stated, because facts can only take individuals as arguments, not actions (or relations or functions).

Meta-level predicates are:

- *Agent-Needed*: this predicate links an agent that can perform a task to an action, e.g., *(Agent-Needed Ticket-Selling-Agent Buy-Ticket)*
- *My-Facts*: this predicates applies to object-level predicates whose instantiations an agent should not check with other agents, but should instead rely on its own information. The location of that agent and the items it possesses would fall into this category, e.g., *(My-Facts Location)*.
- *Inform*: this applies to predicates about which the agent must ask for information from another agent, and always applies to effects of actions. For example, if an action has a effect:
  (Has-Ticket Me Edinburgh London ?Confirmation-Number), the PA must ask the ticket selling agent to instantiate the confirmation number once the action has been performed.

In addition, the meta-level ontology contains information about the predicate hierarchy. This allows us to declare, for example, that the predicate $US\text{–}Dollars$ is a subclass of the predicate *Dollars*, which is a subclass of the predicate *Money*. A predicate hierarchy allows to represent information about how predicates are related to one another.

These two different levels can either be represented as two different ontologies, or the meta-level can be flattened so that all the information is contained within a single

ontology. Since meta information is not allowed in Ontolingua ontologies, we create a different ontology to contain it.

The predicates above are those which are needed by a PA. A service-providing agent would also require the following information:

- *Ask-Planning-Agent*: this predicate applies to an individual of class predicate if the agent is a service-providing agent, and indicates which facts it needs to check with the agent that is requesting an action to be performed.

- *Ask-Other-Agent*: this applies to predicates that need to be checked with another agent, other than the PA or the service-providing agent, and must contain information about which agent it should be checked with.

It is reasonable to assume that any agent that is capable of executing plans through interacting with other agents will have some kind of meta-information about how to find an agent to perform the task. This may be, as we have assumed here, through simply knowing where to find the right agent, or it may be through the ability to get in contact with some kind of brokering service. Additionally, it is reasonable to assume that such an agent will have some way of knowing how to interpret the effects of such an action; when it needs to request further information, and so on.

Although we could not expect agents not designed by us always to represent this information in precisely the manner we have envisaged, we can be confident that this information is represented somehow, and thus these assumptions do not limit the applicability of the system. It may be, however, that if an agent that used a different representation for this meta-information wished to make use of the system, some adaptation may be necessary.

4.5.2 PDDL

PDDL (Planning Domain Definition Language) was chosen as the planning representation because it is the most commonly used planning formalism, and the majority of efficient modern planners take input in PDDL. It is the language developed by the AIPS-98 Competition Committee for use in defining problem domains, and is a community standard for the representation and exchange of planning domain models [Fox and Long, 2003]. The PDDL we use is naturally constrained, because the full functionality of PDDL is not required to express our KIF ontologies. Some of the more advanced features of PDDL, such as the ability to deal with durative actions, are not required in our system.

4.5.2.1 Expressive Differences Between PDDL and KIF

There are important expressive differences between KIF and PDDL. KIF is full first-order, whereas PDDL is used by propositional planners, and hence cannot be full first-order. In fact, the very reason that PDDL is so suitable for planning is that it is not full first-order, and thus does not face the huge search problems associated with first-order logic. However, because it is much more intuitive and convenient to represent ontologies in first-order logic, PDDL is designed as a pseudo first-order language. PDDL appears to be first-order: predicates can be expressed in general terms, with class restrictions on the arguments, rather than simply by a list of all current instantiations of this predicate. However, this is, in fact, merely syntactic sugar. PDDL is designed to look like a first-order language, but to be immediately translatable by planners into a propositional language, and this places restraints on what can be expressed in a PDDL ontology. Thus the expressiveness of PDDL is not equal to the expressiveness of KIF, and there are statements that can represented in KIF that cannot be represented in PDDL.

One problem is that quantification over infinite domains can be represented in KIF but not in PDDL. It would be possible to partially translate KIF ontologies that contain quantification over infinite domains into PDDL so that plans could be formed from them, but the PDDL representation would have to use a particular instantiation of the quantification, and thus the full meaning of the KIF representation would be lost. However, this is an inherent problem when attempting to form plans from such a representation; it is, in most cases, not tractable to plan with infinite domains, and thus whatever representation was chosen for planning, this issue would need to be addressed. In the approach that we take, it is not an insurmountable problem if the KIF and PDDL representations are not equivalent, as long as what is represented in PDDL is also true in the KIF representation. Thus there may be additional information contained in the KIF representation that is not expressed in the PDDL representation, and we would still consider this translation acceptable for our purposes, because the PDDL would only produce plans that were executable both according to the PDDL representation and the KIF representation. The disadvantage is that, because the KIF representation may not be fully expressed in the PDDL representation, there may be valid plans that can be formed from the KIF representation that cannot be formed from the PDDL representation. The loss of expressivity is in one direction only: the PDDL version may not fully express everything that is contained in the KIF version, but the KIF version is certain to fully express everything that is contained in the PDDL version. Thus we can be sure that the plans produced from the PDDL version of the ontology are always executable with respect to the KIF ontology.

The problem of the potential loss of expressivity in the translation to PDDL is difficult to avoid. The very reason that we wish to plan with PDDL rather than directly from KIF or some other first-order representation is because the limited expressivity of PDDL makes it suitable for planning. We thus believe it is reasonable to accept this limitation.

4.5.3 Prolog

Since we required a first-order representation for the ontology, and since the agent communication was written in Sicstus Prolog, an ontology written in Prolog is both sufficiently expressive and easy to read.

4.5.3.1 Expressiveness in Horn Clause Representation

The Prolog programming language is built on *Horn clauses*, with a built-in unification algorithm. A clause (a disjunction of literals) is a Horn clause if it contains at most one positive literal, which means there must be at most one literal on the right-hand side of the implication . A Horn clause with a positive literal is called a *definite clause*, and one without is called a *goal*. Only a subset of first-order logic can be expressed as Horn clauses, and thus it is theoretically possible to develop an ontology in a full first-order representation such as KIF that cannot be fully expressed in a Prolog representation. In particular, it is not possible to express a disjunction of literals on the right hand side of the equation.

However, in ORS we are not representing the KIF action rules as Prolog rules, but rather we are using Prolog as a meta-language in which to encode the KIF rules. If Prolog is used in this way then it becomes possible to use it to express full first-order logic. In the Prolog representation of the ontology, a rule, named *This Rule*, would be expressed as follows:

$[This\ Rule, [A_1, A_2], [B_1, B_2, B_3]]$

Thus, if there were a disjunction in the right-hand side, this could be expressed:

$[This\ Rule, [A_1, A_2], [B_1, or(B_2, B_3)]]$.

Another problem is that quantifiers are not explicitly allowed in Prolog. Thus an expression containing quantifiers must be skolemised. However, we are not dealing with

ontologies that contain quantifiers, and thus we do not need to concern ourselves with such issues.

Thus there is no loss of expressivity in translating between our KIF ontologies and the Prolog representation of this.

Another difference between Prolog and KIF is that in Prolog, there is no difference between a correct, negative answer and a failure of the program to respond correctly to the query: both receive a *no* response. However, in our KIF ontologies, we are not using the closed world assumption, and thus there is a difference between a negative response and an inability to answer the query. In practice, this is not a problem for the system, because, although the agents are written in Prolog, the responses they give to queries are not directly the results of executing Prolog programs, but an interpretation of the failure or success of the Prolog. For example, an attempt to find the answer to a query may result in a *no* response from the Prolog, and, depending on from where and under what circumstances this *no* was returned, this might lead to, for example, a response from the agent of *I cannot correctly answer that query*, or it might lead to a response of *I do not understand the question* . Since it is possible to program the agents to be able to distinguish between the different kinds of responses, it is possible to get them to give appropriate responses to all queries.

4.6 Rewinding the Past

We discuss in Chapter 5 how there are some situations in which we cannot be certain that we are performing the correct refinement. Also, refinements are performed to bring the PA's ontology in line with the ontology of the service-providing agent to which it is currently communicating. Later communication with other agents may indicate that it was unwise to attempt to align with this agent: perhaps it is an anomalous agent (see Section 8.4.3 for further details of this). It is therefore important to keep

a record of the refinements that have been performed thus far. A more sophisticated approach to this issue is to keep several versions of ontologies, rather than one central ontology to which all the refinements are made: this approach is discussed further in Section 8.3. In the current system, we simplify this issue by simply returning a list of all the refinements that have been performed. It is usually not difficult to undo refinements, because, in most cases, the refinements have inverses: for example, adding an argument to a predicate is undone by removing it; switching the order of arguments in a predicate is undone by switching them back again. However, in situations where performing a refinement involves removing information, then, although performing the inverse refinement is possible, it will lead to a situation where there is less information in the ontology than previously. For example, if a predicate is altered from *(Money ?Agent ?Amount ?Currency)* to *(Money ?Agent ?Amount)*, then the fact *(Money Agent1 100 Dollars)* will be altered to *(Money Agent1 100)*. If this is then undone, then the definition of the predicate can be altered appropriately, but the fact *(Money Agent1 100)* can only be altered to *(Money Agent1 100 ?Currency)*, because the specific information has been lost.

As well as altering due to refinement, the ontology is also altering as a result of the actions being performed during plan execution. These effects are only concerned with the facts of the ontology: facts are added and removed because they are part of the effects of an action. Such changes will never influence the representational language of the ontology. It is never possible to rewind the past with respect to the effects of plan execution as it would be with respect to refinement. This is because the actions taken during plan execution have actually happened: they have affected a world larger than the PA. If the PA has executed a *destroy-file* action by requesting a *file destroying agent* to perform this task for him, it then becomes impossible for the file concerned not to have been deleted, and therefore pointless to consider what would have happened had that not been the case. If the PA wishes to rewind the past with respect to plan execution, the only option is to form a new plan from the position in which the PA is

now in to take it back to its original position. In many cases, such as in the example given above, this may prove to be impossible. If the system were designed so as to have multiple ontologies, as discussed in Section 8.3 and above, then they would all need to be altered according to the effects of plan execution. Whether a refinement should have been performed or not is sometimes uncertain and may differ with respect to different agents, and thus we wish to retain the ability to undo them. Whether a plan execution step should have been performed or not is irrelevant: it has been performed, and it is not afterwards possible to undo it.

4.7 Summary

This chapter introduces ORS, which is discussed in more detail in Chapters 5 and 6. The implementation of our ideas has forced us to decide on an exact context for the theory to be used. For reasons discussed in 4.1.3, this was chosen to be a planning context. In addition, the production of such a system has forced us to make many decisions as to how to contain the system sufficiently so that a fully working version can be produced within the time scale. This has led to the production of a system that is not as comprehensive as we would like; this is inevitable when tackling such a vast problem from a novel direction.

The system that we have produced is capable of:

1. Starting with a KIF ontology and a goal written in Prolog;
2. Translating the ontology and goal into:
   (a) PDDL,
   (b) Prolog;

3. Sending the PDDL ontology to a planner, receiving and interpreting a plan and translating it into Prolog;

4. Deconstructing the plan using the Prolog ontology, to produce a justified plan;

5. Executing the plan through communication with other agents;

6. In case of plan execution failure, diagnosing what ontological mismatch between the PA and the action-performing agent caused the failure;

7. Refining the KIF ontology appropriately so that this mismatch will no longer cause problems;

8. Replanning using the updated ontology and repeating the process until the goal is reached or until replanning, diagnosis or refinement is impossible.

We claim that points 2, 4, 6 and 7 represent original research, whilst the rest of the functionality of the system is necessary to provide a platform from which to implement this original research. Of these, points 6 and 7 represent the central focus of the project, and points 2 and 4 are part of the platform from which this occurs.

Chapter 5 discusses the central concerns of this project: the diagnostic and refinement aspects of this system. Chapter 6 discusses the other subsystems of ORS.

Chapter 7 describes the results of running different ontologies, with different ontological mismatches, on the system, and the steps that we took to evaluate both the functionality and the usefulness of the system through comparison with ontological mismatches discovered in real-world ontologies.

A discussion of how we would like to improve the system, and our ideas as to how this could be achieved, can be found in Chapter 8. Nevertheless, the produced system fulfils all the requirements of the specification, and acts of a proof-of-concept for dynamic ontology refinement.

Chapter 5

Diagnosis and Refinement

5.1 Introduction

The diagnostic and refinement processes are implemented as separate systems, but the theories behind them are deeply intertwined. The heart of the project is the theory on which these two systems are based.

ORS must first identify where the ontological mismatch lies: diagnosis; secondly, it must apply methods to fix this problem: refinement. The ability of the system to perform either of these tasks depends on the pre-identification of the kinds of mismatches that could occur. Before the systems were implemented, substantial ground work was necessary to formalise the ontological mismatches that are possible for the given ontological representation. The efficacy of the system depends to a large extent on our ability to define refinement techniques and to use them to identify problem areas and hence choose an appropriate refinement technique to apply.

The choice of ontological representation predetermines the kinds of ontological mismatches that are possible. In this project, we are interested in first-order representations. The method of identifying what mismatches could occur and forming diagnostic

algorithms to determine which of these is to blame for communication failure is general to any ontological representation; however, the types of mismatches that could occur and the algorithms needed to chose between them will vary between different representations. This issue is discussed in more detail in Section 8.1.2.

5.2 Types of Refinements

In order to apply reasoning techniques to the problem of ontology mismatch, we need to define methodical descriptions of how we might expect ontologies to differ. It is not enough merely to say this ontological object is different to that one, we also need to say how they are different. Therefore, we have developed formal descriptions of how first-order ontologies may differ from one another. These descriptions of how things may differ can be transformed into techniques which describe how refinements that patch these mismatches can be implemented.

The system is limited in that we can only diagnose and refine ontological mismatches that correspond to these descriptions. For ontological objects that differ in a more chaotic manner, or in a manner not considered by the diagnostic system, no appropriate diagnosis will be discovered, and thus no refinement technique will be applicable. This is an inherent limitation of the system: no matter how thorough we are in the investigation of ontological mismatches, it is always conceivable that ontological mismatches exist that have not been described. This makes it difficult to deal with ontological differences that are chaotic in nature: for example, names that are arbitrarily changed, arguments that are arbitrarily altered, and so on, we cannot always expect to find appropriate refinements using descriptions of methodical differences. Dealing with such problems is outside the scope of this project. In order to limit the number of problems of the latter kind; that is, problems that differ in a methodical way but for which we have not predefined diagnostic methods, we need to ensure that we are as

thorough as possible when defining these methods.

Many of the possible refinements fall into the category of those that generalise an ontology, or those that specialise an ontology: often, ontologies are altered because the level of information they contain is inappropriate for their needs. Much work has been done in the field of generalising theories through the techniques of *abstraction*, and it is to these that we turn as a basis for developing our own techniques. Although much less work has been done in the field of specialising theories, we can apply these abstractions in reverse to create what we call *anti-abstractions*, which are thus formal descriptions of how to add detail to a theory.

5.2.1 Abstraction

The problem of refinement that results in generalising and removing detail from a theory is the problem of abstraction. This is frequently used in automated systems to detect patterns and remove detail in order to create a more general, simpler plan [Giunchiglia and Walsh, 1992]. Thus we examine abstraction techniques to draw inspiration for our own refinement techniques.

In [Giunchiglia and Walsh, 1990], Walsh and Giunchiglia attempt to formalise the hitherto vague notion of abstraction by creating a theory of abstraction. They claim that the most common use of abstraction in theorem proving, problem solving and planning has been to abstract the goal, to prove its abstracted version and then to use the structure of the resulting proof to help construct the proof of the original goal. This work includes some work on how to build abstractions. Since abstractions should change the theory and not the logic (except in the case of reducing a unary predicate to a nullary predicate, which could be seen as reducing first-order logic to propositional logic), the problem of building abstractions can be reduced to the problem of deciding on a suitable mapping of atomic formulae. It is also desirable that these abstractions

be *truthful*, where a truthful abstraction is one such that all the theorems in the ground space are also theorems in the abstract space. They further claim that most abstractions of this sort fall into four categories:

1. **Predicate abstractions**

   mapping predicate names in some uniform way:

   e.g., *(Bottle ?X)*, *(Cup ?X)* map onto *(Container ?X)*.

2. **Domain abstractions**

   mapping constants and function symbols in some uniform way:

   e.g., *(Prime 3)*, *(Prime 5)* map onto *(Prime Oddnumber)*.

3. **Propositional abstractions**

   dropping some or all of the arguments to predicates:

   e.g., *(Abelian GroupA)*, *(Abelian GroupB)* map onto *(Abelian)*.

4. **Precondition abstractions**

   mapping some of the atomic formulas onto true or false:

   e.g., *(Has Ticket Me)* $\rightarrow$ *(Can-Travel Me)* maps onto *(Can-Travel Me)*

5.2.2 Anti-abstraction

When we find that our ontology contains too much information, that we have a representation of the domain that is too specific, we will want to abstract detail from our ontology. When we face the opposite situation; that our ontology is not detailed enough, we need to perform the opposite task: adding detail to the ontology. This situation seems to be fairly likely to occur. An ontology is likely to become increasingly sophisticated over time, with further detail added as further work is done in the domain, and thus an agent with a slightly outdated version of this ontology will need to be able

to add this information to its ontology. Additionally, an agent working in an area outwith its usual field of expertise may have somewhat vague ideas about this area. It may not have been considered worthwhile to provide a rich and detailed description of all the information in this field when the agent was not expected to interact significantly with it. This is particularly relevant in large and complex domains where developing a complete ontology is an onerous task; developing as much of an ontology as is initially thought to be relevant and then allowing the agent to insert more detail as and when it proves necessary is perhaps a more time-efficient option. This has interesting parallels to human reasoning, where our ideas about domains in which we are not knowledgeable are hazy and generalised. As we interact further with this domain we enrich our understanding of it so that it becomes more complete in a fairly similar manner to how an agent might enrich its ontology using the anti-abstraction techniques.

The refinements below illustrate how the abstractions above can be inverted to produce methods that can be used to insert detail into an ontology. Note that we use infix notation, with the first element of a predicate referring to the predicate name, and the remaining elements, to the predicate arguments.

1. **Predicate anti-abstraction**

   A single predicate is split into one or more subclass predicates, e.g.,:

   (Money ?Amount) maps onto *(Dollars ?Amount), (Euros ?Amount), (Sterling ?Amount)*, etc

2. **Domain anti-abstraction**

   The class of an argument is divided into one or more subclasses, e.g.,:

   (Money ?Amount European) maps onto *(Money ?Amount Euros), (Money ?Amount Sterling), (Money ?Amount Krona), etc.*

3. **Propositional anti-abstraction**

   The arity of a predicate is increased, e.g.,:

(Money ?Amount) maps onto *(Money ?Amount Dollars)*, *(Money ?Amount Sterling)*, *etc.*.

4. **Precondition anti-abstraction**

   A precondition is added to a rule, e.g.,:

   (Has Money ?Agent) $\rightarrow$ *(Has Item ?Agent)* maps onto

   (Has Money ?Agent) $\wedge$ *(InStock Item Shop)* $\rightarrow$ *(Has Item ?Agent)*

5.2.3 Other Types of Refinement

There are only a finite number of ways in which the kind of ontology we are considering can be altered. Such ontologies contain the following objects:

- Predicates, which have a name, an arity and a given class for each argument;
- Action rules, which have one or more preconditions and one or more effects;
- Classes, which have a name and a superclass (the set of classes and their superclasses in the ontology defines the class hierarchy);
- Individuals, which have names and classes;
- Facts, which are applications of predicates to individuals.

Therefore, the only possible way to change such an ontology is to change one of these attributes. Of these, the predicates and the classes are signature objects, which make up the representational language, and the action rules, individuals and facts are theory objects, which are expressed using the representation language.

The abstractions and anti-abstractions above describe specific ways of changing predicates and actions rules. However, they do not cover all possible ways of changing these objects, nor do they describe how the other ontological objects could be changed. In

this section, we examine each kind of ontological object and discuss all the possible ways it could be changed. Note that we have throughout made the assumption that changes are made one at a time and that we are not dealing with compound mismatches. A discussion about how we might do this can be found in Section 8.4.4.

- **Predicates:**
  - **Changing a predicate name:** We have already explored two ways in which this might be done: by making the name more specific (predicate anti-abstraction) and by making the name more general (predicate abstraction). In both these cases it is possible to deduce this change of name because we can relate the existing name to the correct name, since we have information that it is a sub- or super-predicate. If this name is changed arbitrarily, then it is not possible to diagnose the problem, as the new, arbitrary, name cannot be connected with the existing name. There may be no way to know what predicate this new one is linked to if we cannot find a connection, although it is possible that there may be some information regarding synonyms in the ontology.
  - **Changing the arity:** Any possible way of doing this is accounted for by propositional abstraction or anti-abstraction. Although these techniques deal with removing or adding a single argument, they can be repeated as necessary to deal with removing or adding any number of arguments.
  - **Changing the classes of arguments:** If the classes of the arguments are changed to a subclass or superclass, this is accounted for by domain abstraction or anti-abstraction. If this is done in a less methodical manner, it is still not difficult to diagnose the appropriate refinement. Since the predicate still has the same name, it is simple to link the new occurrence of the predicate to the existing one, thereby determining which predicate

should be altered. The class of the new argument must then be found, either through examination of the ontology or, if this fails, questioning of the other agent.

- **Switching the classes of arguments:** Another possibility is that the arguments are transposed; this is also simple to diagnose.

- **Action Rules:**

  - **Changing preconditions:** Some ways of doing this is described by precondition abstraction and anti-abstraction. It is also possible that an existing precondition is altered, although this can be seen as being equivalent to performing first an abstraction and then an anti-abstraction (or vice-versa).

    An interesting way in which preconditions can be added or altered is tightening class restrictions. Normally, the class is restricted through the predicates used: for example *(HasPaper ?Agent ?Paper)* restricts the second argument to being of class *Paper*. For some rules, however, this may not be sufficient. Perhaps a *submitPaper* action requires a paper to be of a certain format. If *(HasPaper ?Agent ?Paper)* is the only precondition, then an extra class restriction must be added to the rule to enforce this restriction: e.g., *(Class Paper PdfPaper)*. Altering these preconditions is another kind of refinement.

  - **Changing effects:** Changing the effects of an action occurs in precisely the same manner as changing the preconditions.

- **Classes:**

  - **Changing the name:** This is extremely difficult to deal with, because it is usually impossible to know what existing class this new class is replacing. This new class can be added into the class hierarchy if information is known about the superclass, but it is usually impossible to replace the existing

class with the new class.

- **Changing the superclass:** If the name of the new superclass can be identified, changing the class hierarchy in this manner is not difficult.

- **Individuals:**

  - Changing individuals merely consists of changing their names. This is not an interesting change. It is also difficult to deal with, for the same reasons as changing the name of a class or a predicate is.

- **Facts:**

  - If a fact is changed, it is sometimes possible, and sometimes not, to determine what the new version of the fact should be. Determining whether a fact should be changed, and how this will affect the knowledge base, is usually a matter of belief revision.

In addition to changing ontological objects that are already present in the ontology, whole ontological objects can be added or removed. It is difficult to deal with objects that are added unless we can extract sufficient information about how they are related to objects that already exist. A class, for example, can be added fairly easily, as the only required information is the name and the superclass. For a predicate to be added, we firstly need to know how the predicate is related to other predicates. It is also necessary to know the arity and the classes of each argument (which may involve adding new classes). This is possible in some communication situations, and not in others. An action rule cannot be added without full information about its preconditions and effects. Removing objects from an ontology is not difficult to perform, although it is important to ascertain that the object should be entirely removed from the ontology, rather than removing specific instances of it (for example, removing a particular predicate as a precondition of an action rule, but retaining the predicate definition and other

occurrences of the predicate). However, this is difficult to diagnose, because unnecessary additional objects in the ontology would not normally lead to plan execution failure.

Those alterations that occur in a methodical manner, or for which it is possible to connect the object that has been changed with the object from which it has been changed, can usually be accurately diagnosed. Those alterations that are chaotic and hard to link to the original ontological object can usually not be accurately diagnosed. In Section 5.3, we describe the diagnostic process, making it clear exactly which changes in which circumstances we can diagnose, and what can be done if precise diagnosis is not possible.

5.3 The Diagnostic Algorithm

Having outlined the ways in which ontological mismatch may occur, we now describe how we can use this knowledge to diagnose the specific mismatches we encounter.

5.3.1 Diagnostic Assumptions

As discussed in Section 4.1.2, making the problem of ontology refinement tractable in the first instance requires some assumptions to be made. Here, we outline what these assumptions are.

- It is an assumption in the current system that external agents are only interacting with the PA, and not interacting with others in ways that cause changes to their ontologies. Supposing we have, at some stage of the plan execution, interacted with a particular agent, say AgentX. After that interaction, AgentX will have certain beliefs about the state of the world that may have been affected by the

interaction: for example, if it has performed an action for the PA, it will believe that the effects for that action are fulfilled. We are making the assumption that if, at some later point in the plan execution, we wish to return to AgentX and question it about, for example, its beliefs about the effects of this action, then its beliefs will be the same as just after that action was performed. If AgentX had been interacting independently with other agents then this might not be the case; any questions we ask it about the effects may not reflect the state of the effects immediately after the action was performed, and thus cannot tell us anything certain about the action. This assumption is certainly not valid in all cases, and an obvious way to make the system more sophisticated is to eradicate such an assumption. However, this enables us to develop a system that can explore these issues in a context that is simple enough for a working system to be produced.

- We make certain assumptions about the way in which a service-providing agent will check its preconditions. There are four categories of preconditions, which are checked in the following order:

  1. Those that the service-providing agent can check independently; for example, for a *shopkeeper agent*, *(InStock Item Shop)* would be a precondition that it could verify itself.

  2. Those that must be checked with an agent other than a PA agent. For example, a *ticket-booking agent* might check with a *airline agent* to check if a requested ticket was actually available.

  3. Those that must be checked with the PA; for example, *(Money Agent Amount)* would be a precondition it would need to check with the PA.

  4. Those that can verified independently, but only after information has been received; for example, *Price* $<$ *Amount* would be something the shopkeeper agent could verify independently once it has received information about *Amount* from the PA.

This ordering simplifies the process of diagnosis. If the service-providing agent begins to query the PA, and failure then occurred, it can be assumed that the cause of failure is the reply to one of these questions. Without this assumption, we cannot be sure whether failure is due to the reply to a question, or another precondition that was being checked in the meantime without being put to the PA. If we wished to remove this assumption, the diagnostic process would have to be complicated to some extent to cover this possibility. However, this would only affect a small number of situations. This is discussed further in Section 5.3.3.

- The PA always interacts with the first service-providing agent it can find that it believes is able to perform the task. In cases where the PA is aware of more than one agent that it believes can perform the task, there is no deliberation over which agent would be more appropriate to approach; the choice depends only on the order in which they appear in the meta-ontology. However, it would be preferable to make this choice in a more sophisticated manner. For example, previous interactions with the agents could lead to one being considered more reliable than the other; discussed in Section 8.4.3. Additionally, one of the agents may be more flexible and willing to accommodate the PA, making it a more desirable agent to interact with; this is discussed in Section 8.4.2.

- The system is designed to deal with errors on a case by case basis. We do not need to make the assumption that failure is caused by exactly one error; we can identify one error, fix it, replan, encounter failure again and then diagnose a second error. This is not the most efficient way to deal with such a situation, but is reasonable if we believe that in most cases, a particular failure will be caused by a particular error. However, the nature of this approach to multiple errors means that we are forced to assume that these errors are independent, and that dealing with them one by one will lead towards a more correct ontology,

rather than refining the ontology in an incorrect manner. This is certainly not always going to be a valid assumption in a real world situation, and the system will fail if it encounters compound mismatches that cannot be diagnosed as a series of individual mismatches. Exploration of how we might investigate these combinations of errors would add a layer of sophistication to the system and enable it to be more robust in complex domains. This is discussed further in Section 8.4.4.

- We assume, for reasons discussed in Section 6.2, that the only information we can get from other agents is that which is revealed by the questions they put to the PA, and by their answers to direct questions put by the PA to them.

5.3.2 Determining Authority

When an ontological mismatch is detected, there is the potential for some complex negotiations between the agents as to which of them ought to refine their ontologies, or, indeed, whether both should. Factors that might be relevant here are whether one of the agents is recognised, perhaps by the community, perhaps by the other agent, to be an authority on the matter; whether one is controlling the situation by, for example, being able to provide something that is required by the other agent; whether either of the agents consider this part of their ontology to be particularly important and are unwilling to compromise it, and so on.

We simplify all these issues by assuming that the PA is willing to take on trust any information that is given to it by another agent. The main reason we make this assumption is that it makes the situation tractable and allows us to avoid getting sidetracked onto important but tangential issues. However, we also believe that this assumption is plausible, because, in this scenario, the PA is interacting with other agents because it wishes them to provide services for it; hence the other agents are in control of the

situation. We discuss how this could be made more sophisticated in Section 8.4.2.

5.3.3 Linking Plan Failure to Ontological Mismatches

The means of detecting that an ontological mismatch has occurred, and determining what that ontological mismatch may be, is through agent communication. The only information available about mismatches is that which can be gleaned from observation of past agent communication, from forming appropriate questions and putting them to the appropriate agent, and from analysis of the ontology.

The questions put to the PA by the service-providing agents provide the richest source of information. In many cases, the source of the ontological mismatch can be directly identified from these questions, through the information these questions reveal about the ontology of the service-providing agent.

We have developed the notion of *surprising questions*, which can provide information about where ontologies between two agents may differ. The PA will have a set of preconditions for the action to be performed which it believes are all fulfilled at the time that the action is to be performed. Some of these it will expect to be asked about and some not; however, it will not expect to be asked about anything not directly contained in these preconditions. When a question is put to it by the service providing agent, it answers it as best as it can and then compares this question against its list of preconditions to see if it exactly matches one of these. If it does not, it is flagged as a surprising question. No further action is taken at this stage, but if plan failure occurs then the surprising questions are referred to. Of particular interest are surprising questions that are asked immediately before plan failure occurs.

Very often, the information contained in a surprising question is enough on its own to find the source of the problem. For example, if the question contains an instance of a recognised predicate, but with an unexpected arity, or with an argument with an

unexpected class, then it is clear that the expectations of this predicate are to blame. However, sometimes it is discovered, possibly through information revealed in a surprising question, that there is a problem with a fact, believed by the PA to be true, but believed by the service-providing agent to be false. If there is no problem with this fact with respect to the representational language, linking the plan execution failure to a flaw in the underlying ontology is more difficult. It must be established how the fact came to be believed. The fact may be present in the original ontology, or it may have been added to the ontology because it was believed to have been the effect of a previously performed action. In the former case, our policy is to remove the fact from the ontology, preferring the service-providing agent's belief that it is incorrect to our own belief that it is correct. In the latter case, we must examine the faulty action to discover what went wrong. We refer to the algorithm that diagnoses these incorrect facts as the Shapiro algorithm, because it is loosely inspired by Shapiro's work on algorithmic program debugging [Shapiro, 1982] (see Section 3.3.5). We have not attempted to follow Shapiro's algorithms closely but have merely used his ideas as an inspiration.

The algorithms used in the diagnostic processes are illustrated below as flow charts. Figure 5.1 illustrates the top-level decision procedure, and Figures 5.2 - 5.7 illustrate sub-algorithms that are called once the higher-level decisions have been made.

Figure 5.1 separates the diagnosis into three separate cases: those where no questions were asked; those where questions were asked but none of them were surprising; those where there were questions asked and at least one of them was surprising.

1. **Failure immediately after a request to perform an action has been made**

   This situation is illustrated in Figure 5.2.

   In this situation, it can be difficult or even impossible to diagnose what the cause of failure is because the amount of information we have access to is quite limited. The fact that failure has occurred without any questioning is certainly helpful

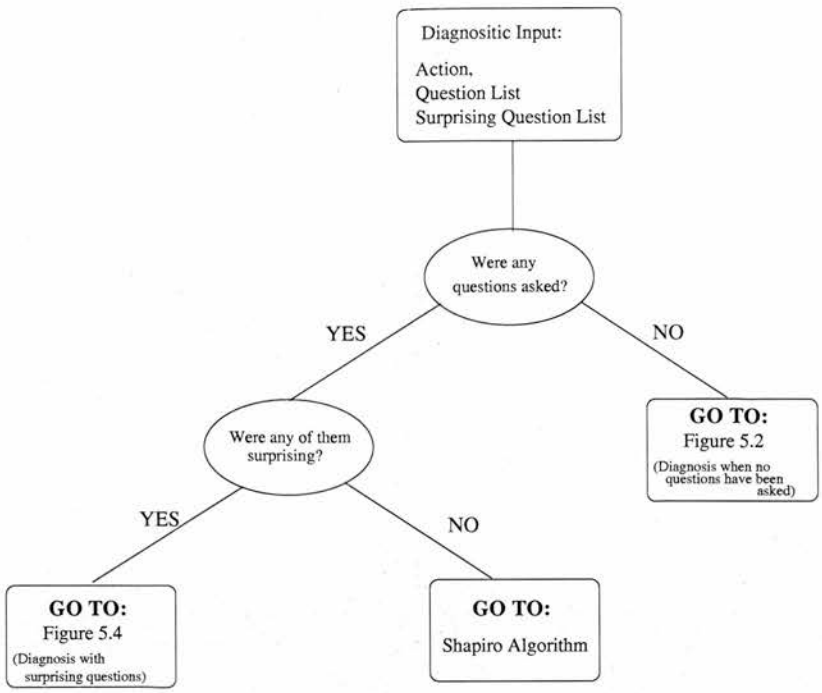

Figure 5.1: Top Level Decision Making

information, but it reveals less about the ontology of the other agent than the other two situations.

We identify three situations in which this may occur:

- The agent that has been asked to perform a task it is not able to;
- One of the preconditions for the action, that we believed to be true, is not true;
- We are missing a precondition, and that precondition is not currently fulfilled.

We can eliminate the first option by querying the agent as to whether it is capable of performing the action. If not, we must remove the information that this agent can perform the action from our ontology and replan. Note that this means a change to the meta-ontology, rather than to the ordinary ontology, as this is where information about which agent can perform which task is kept.

We can investigate the second option by querying the other agent as to its beliefs

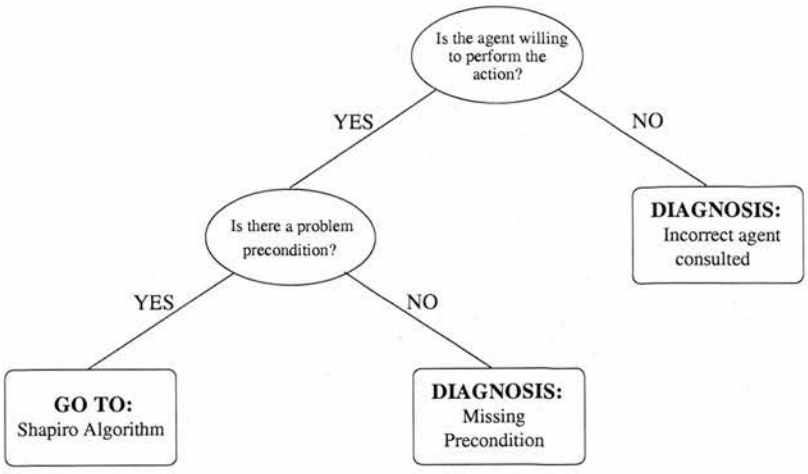

Figure 5.2: Diagnosis when no Questions have been Asked

of the truth values of our preconditions. If we discover a precondition that it does not believe to be true, then this is a likely cause of the failure. We then need to consider why we came, incorrectly, to believe this precondition to be true by using the Shapiro algorithm, illustrated in Figure 5.3.

If all of the PA's preconditions are fulfilled then the problem is much harder to diagnose. It is clear that not all of the other agent's preconditions were fulfilled, since it has refused to perform a task that it is capable of performing. Since it believes that the PA's preconditions for the action are fulfilled, it is clear that there must be some discrepancy between the two sets of preconditions: there is an essential precondition for this action that the planning agent does not have in its set of preconditions. Unfortunately, we have no information at all about what this precondition may be. We cannot glean any information from previous queries, since there are none that relate to this action. Nor can we query the agent about what this precondition is, since we have no basis to guess what it might be, and we cannot assume that the other agent will reveal its complete set of preconditions for this action. Thus our only recourse during refinement is to flag this rule as incomplete and not use it during planning. Hopefully, a new plan can be formed that does not require the use of this incomplete rule.

However, if this rule is essential to forming a plan that can achieve the goal then we find ourselves in a situation where it is impossible to form a sound plan. This is naturally undesirable; however, it is still preferable to our former state of ignorance, where we believed that we could form an executable plan and only discovered during execution that this was not the case.

Thus there are four possible diagnoses in the situation where failure occurs without any questions being asked:

(a) Incorrect agent consulted;

(b) An effect for a previous rule should be altered;

(c) A precondition should be altered;

(d) Missing or incorrect precondition. In this situation, precise diagnosis is not possible. See Section 5.4.

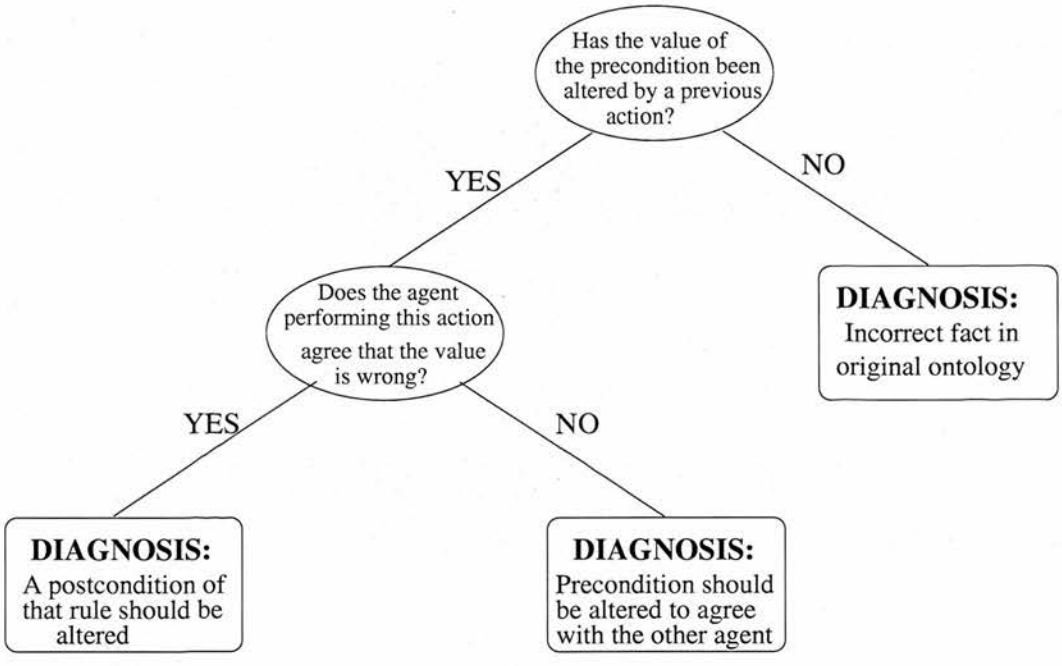

Figure 5.3: Shapiro Algorithm Diagnosis

2. **Failure after a surprising question** [1]

[1]Note that *RSQ* refers to the Relevant Surprising Question

This situation is illustrated in Figure 5.4.

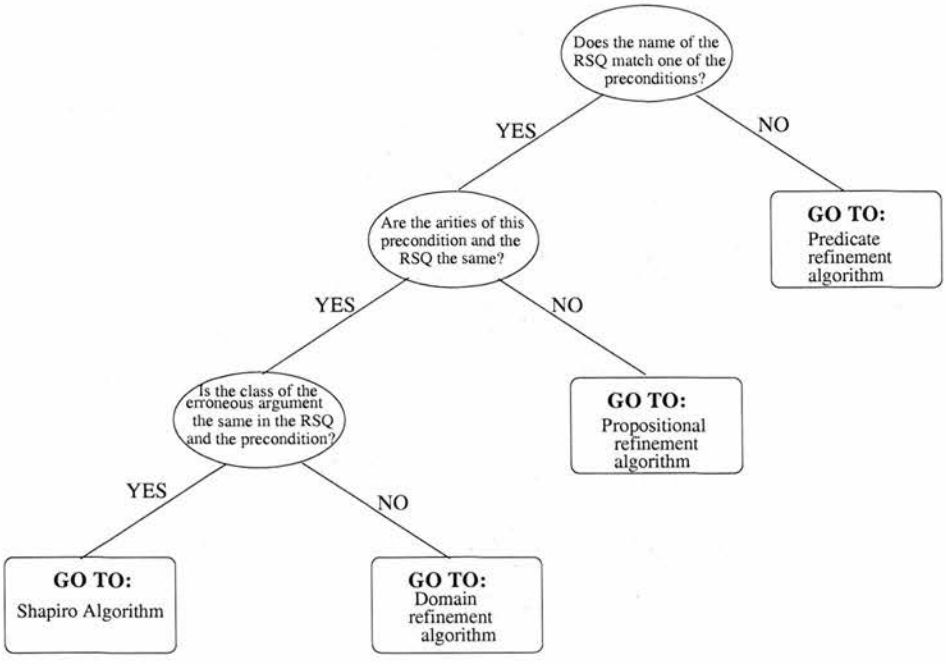

Figure 5.4: Diagnosis with Surprising Questions

If the agent with whom the PA is conversing attempts to perform the action, and fails without any questioning of the agent, then the information available for diagnosis is somewhat limited. However, if any queries are made, we are immediately provided with much information about what the problem may be, particularly in the case where the question is surprising. A surprising question is one that does not *exactly* match a precondition; it may be very similar to an expected precondition.

We can categorise surprising questions in the following ways:

(a) The name of the predicate in the surprising question matches the name of a precondition:

   i. The number of arguments of these two 'matching' predicates is the same: **domain refinement** or **switch arguments**;

   ii. The number of arguments is different: **propositional refinement**;

(b) The names do not match:

i. There is a class relation between the name of the predicate in the surprising question and the name of one of the preconditions: **predicate refinement**;

ii. There is no relation: **precondition refinement**.

- **Case a)i):**
  - Most of these situations conform to the following [2]:

    $(f\,\vec{x}\,y)$ matches $(f\,\vec{x}\,z), y \neq z$,

    where f indicates a predicate, $\vec{x}$ indicates one or more arguments, and y and z are arguments.

    There are four possibilities:

    * $class(y) = class(z)$

      In this case, the two predicates match with respect to the representational language, but conflict with respect to their specific instantiation. The source of this incorrect fact must be tracked down by the **Shapiro algorithm**.

    * $class(y) \prec class(z)$.

      This is **domain abstraction**.

    * $class(y) \succ class(z)$.

      This is **domain anti-abstraction**.

    * $class(y)$ is not related to $class(z)$.

      This does not fall so neatly into our classifications, but it is nevertheless easy to diagnose and refine, because we have sufficient information about the problem.

[2] Notation:
i) In all cases, the statement "A matches B" indicates that A is the surprising question put by the service-providing agent, and B is the question the PA expected, which corresponds with the PA's ontology. Thus diagnosis and refinement is aiming to change B so that it matches A;
ii) $\prec$ indicates subclass relation;
iii) $(f\,\vec{x}\,y)$ does not imply anything about the ordering of the arguments: it does not imply that y must be the last of the arguments.

$class(z)$ must already be in the ontology of PA, since z is a constant that PA is using in its planning. $class(y)$ may already be known by PA; if not, the service-providing agent is queried to ascertain this value. Since the service-providing agent is using y in its communication, it must know what its class is.

- Sometimes, a mismatch of this type may conform to the following:

  $(f\vec{x}yz)$ matches $(f\vec{x}zy)$,

  or, more generally,

  $(f\vec{x}yz)$ matches $(f\vec{x}ab)$,

  $class(y) = class(b)$ and $class(z) = class(a)$.

  That is, the arguments have been transposed, either with the same instantiation or not.

- **Case a)ii):**

  - $(f\vec{x}y)$ matches $(f\vec{x})$.

    This is **propositional anti-abstraction**. This signature refinement is easy to diagnose and refine; all that is required is that $class(y)$ is ascertained, either through examination of the PA's ontology or through questioning the service-providing agent. However, this signature refinement entails theory refinements that are more difficult to implement; this is discussed in Section 5.5.

  - $(f\vec{x})$ matches $(f\vec{x}y)$.

    This is **propositional abstraction**. This is easy to diagnose and refine, both for the initial signature refinement, and for the theory refinements that entails. It may be necessary to know what the class of the additional argument is, so that it is possible to tell which argument should be removed; this can be discovered either from the ontology or by asking the service-providing agent.

- **Case b)i):**

- $(f\,\vec{x})$ matches $(g\,\vec{x})$

  There are three possibilities:

  * $class(f) \prec class(g)$.

    This is **predicate abstraction**.

  * $class(g) \prec class(f)$.

    This is **predicate anti-abstraction**.

  * $class(f)$ is not related to $class(g)$.

    This is usually impossible to diagnose, because there is insufficient information to determine that $(f\,\vec{x})$ should match precondition $(g\,\vec{x})$. This is much harder to diagnose than the equivalent situation in case a)i), where there was no relation between the classes of the arguments, because in that situation, the matching of the names of the predicates was sufficient information to determine that this precondition probably ought to match the surprising question. In this situation, we do not have this information. Situations of this type are usually incorrectly classified as occurrences of case b)ii): a missing precondition. We cannot connect this surprising question to any of the preconditions, and so we assume that it is an additional precondition.

- **Case b)ii):**

  - $(f\,\vec{x})$ fails to match any of the preconditions.

    In this situation, we diagnose **precondition anti-abstraction**, although, as discussed in case b)i), this will sometimes be incorrect. If it is incorrect, the effect will be to over-constrain the rule. We consider this to be an acceptable approach, because if this does occur, we are still left with a rule that will not be used other than in situations where it is correct to use it. The disadvantage is that there may be some situations in which it is correct to use it in which it appears to be unusable.

However, since we consider that random name changing will not occur especially often, these situations will be rare, and a diagnosis of precondition anti-abstraction is usually correct.

In summary, there are ten possible diagnoses in the situation where failure occurs when surprising questions have been asked:

(a) **Domain abstraction** should be applied (see Figure 5.5);

(b) **Domain anti-abstraction** should be applied (see Figure 5.5);

(c) The arguments should be inverted (see Figure 5.5);

(d) The domain of one of the arguments should be changed in a more chaotic manner (see Figure 5.5);

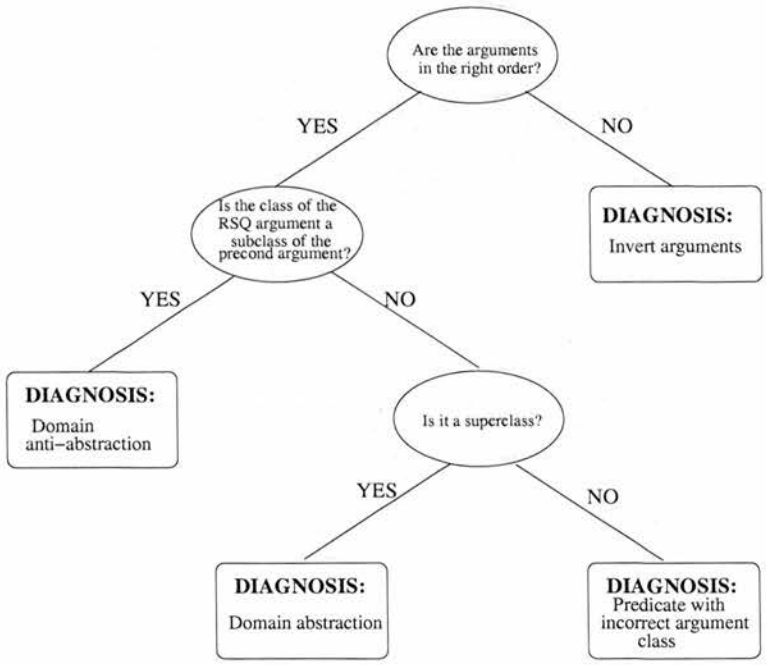

Figure 5.5: Diagnosis of Domain Refinement

(e) A fact in the original ontology should be removed (diagnosed by the **Shapiro algorithm**, see Figure 5.3);

(f) An effect of a previous rule should be altered (diagnosed by the **Shapiro algorithm**, see Figure 5.3);

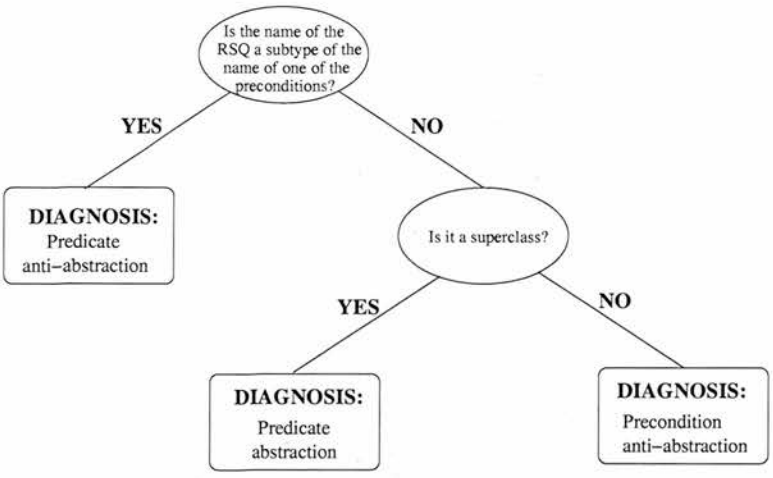

Figure 5.6: Diagnosis of Predicate Refinement

(g) **Predicate anti-abstraction** should be applied (see Figure 5.6);

(h) **Predicate abstraction** should be applied (see Figure 5.6);

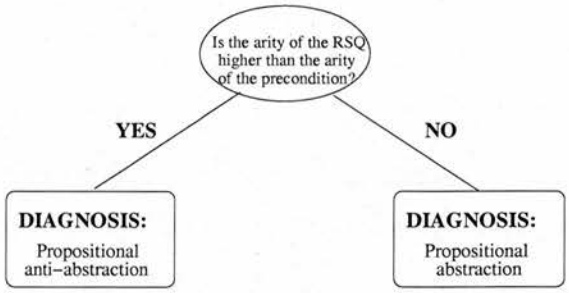

Figure 5.7: Diagnosis of Propositional Refinement

(i) **Propositional anti-abstraction** should be applied (see Figure 5.7);

(j) **Propositional abstraction** should be applied (see Figure 5.7);

(k) **Precondition anti-abstraction** should be applied.

3. **Failure after some questioning, with no surprising questions**

If failure occurs in this situation, we can conclude:

- Since questioning has begun, the only possible cause of failure is the PA's response to these questions.

- The PA was expecting to be asked this question, and therefore ensured that it was correctly fulfilled, as far as it understood this was to be done.

There are two ways in which this might happen:

(a) The service-providing agent put a question to the PA that contained an uninstantiated variable, so that the PA could instantiate this appropriately. For example, the service-providing agent might ask:
(Money PA ?Amount),
to which the PA would reply, for example,
(Money PA 1000).
As part of the preconditions of the action, the PA would have some expectations about what the value of *?Amount* should be; for example, PA might have a precondition:
$?Amount > 800$,
and would ensure that this precondition was fulfilled before the action was attempted.

Failure after PA's reply to the question *(Money PA ?Amount)* indicates the incorrectness of the precondition which moderated the value of this variable: perhaps the correct precondition was
$(?Amount > 1200)$, or
$(?Amount < 800)$.
It is not difficult to diagnose this problem, but it is usually impossible to refine it. Heuristics could be built in to guess what the correct value should be: *i.e.*, guess that we actually need 100 more than the amount stated in the previous precondition, or guess that the inequality should be inverted, and see if such guesses lead to successful execution. However, this is a displeasingly imprecise solution, and in cases where the uninstantiated variable is not numeric, it becomes even harder to guess what the correct value might

be. A neater solution would be to mark the rule as unusable; however, this may severely limit the abilities of the PA.

(b) If the service-providing agent puts a fully instantiated question to the PA, then it is clear that both the service-providing agent and the PA believe that the truth or falsity of this fact is essential to the execution of the rule. The PA must have ensured that the fact was true or false, depending on its expectations of the required truth value, before the rule was executed. Therefore, we can only conclude that PA's expectations of the required truth value were wrong: if PA believe this fact should be true for the action to be performed, it should, in fact, be false, and vice versa. This is both easy to diagnose and to refine.

5.3.4 Dealing with Incorrect Facts: Using the Shapiro Algorithm

In many cases, as has been discussed above, finding a problem fact will enable us to immediately diagnose what the problem is. For example, if we encounter a fact that contains an extra argument than we expect, it is clear that the problem is connected to this argument mismatch. However, in some cases there is no signature error with a problem fact; sometimes the fact is correct from a signature point of view but simply wrong, *i.e.*, incorrect from a theory point of view. This problem is thus not amenable to signature refinement. Instead, we must discover how it was that this incorrect fact came to be believed. The algorithm we use to determine this is the Shapiro algorithm.

In such cases, we need more sophisticated information about how the plan steps are linked to the underlying ontology: which action rules were used to perform each step, why the preconditions of those rules were believed to be correct in each situation, and so on. This information is provided by the plan justification; details of this can be found in Section 6.3.2. If the fact came to be believed because it was believed to be an

effect of a previous action, then we need to establish how the definition of that action is at fault. The most obvious possibility is that one of the supposed effects of the action was erroneously believed to be an effect:

$\Phi \rightarrow \Theta \wedge (f\vec{x})$

should in fact be stated:

$\Phi \rightarrow \Theta$,

where Φ is a set of preconditions, Θ is a set of effects, f is a predicate, $\vec{x}$ one or more variables, and $(f\vec{x})$ represents an extra effect.

This can be checked by questioning the agent that performed this action as to its beliefs about the validity of the effect. Due to assumptions we make about agent behaviour, we can question the agent as to its current beliefs about the truth of the fact and this will represent its beliefs about the truth of this fact immediately after the action was performed. If we find that the agent does not believe this to be an effect for the action, then this effect must be removed from the rule. This may leave the rule incomplete; it may be that this was not an extra effect but rather an incorrectly stated effect. In some cases, we can solve this problem by asking the agent to instantiate the fact for us; that is, we query it using the problem fact with the relevant variable (or all variables if we are uncertain) uninstantiated. In this way, we may be able to discover the correct instantiation for this effect and refine the rule accordingly. Through this method, we can also ascertain what the agent which was performing the action at which the plan failed was expecting as a precondition. Since we have made the assumption that other agents agree with each other, these two instantiated facts ought to agree with each other. If they do not, it is clear that the problem action was not the correct one to employ. This information about the correct effect and the expected precondition will be added to the appropriate rules during refinement, so that a better plan can be developed.

Another possible cause of error is that there was some kind of problem with the preconditions, rather than that the effects are incorrectly stated in the ontology. It is clear

that this problem with the preconditions must have had some affect on the effects, since otherwise it cannot have been the cause of plan failure, but the problem occurs because an incorrect instantiation of the preconditions leads to an incorrect instantiation of the effects, rather than that there is anything inherently wrong with the effects. There are, however, only a limited number of situations in which this can occur, because this problem in the preconditions was not sufficient to cause immediate plan failure. From the fact that this action was included in the plan and was performed successfully, we can deduce that both the PA and the action performing agent believed that the preconditions were fulfilled. So the value of the arguments in the problem fact were sufficient to fulfil the preconditions for the action, but affected the value of the effects. Consider the example given above in failure case 3a) (failure after some questioning, with no surprising questions). In that situation, a reply:

(Money PA 1000) causes failure, perhaps because the precondition:

(?Amount > 800)

ought instead to have been

(?Amount > 1200).

But consider what would happen if the correct precondition was

(?Amount > 900).

In such a case, this misinformation about the precondition would not cause plan failure, because

(Money PA 1000),

would still be an acceptable reply. However, after this action was performed, the PA would use its precondition to deduce that

(Money PA 200)

was now true, whereas, in fact, the truth is

(Money PA 100).

Such situations lead to incorrect facts being believed, which may well affect plan execution later on.

In order to distinguish the two cases, the expected value of the effects is determined by asking the agent that performed the action to instantiate the problem fact. If this error could be accounted for by an error in the preconditions, then this must be investigated further. The Shapiro algorithm is used again to investigate the source of this failure.

This is another area where it may be difficult to tell exactly what the problem is, as there could potentially be a very complex relation between the preconditions and the effects such that an error in the preconditions would result in the effects being altered in an unexpected way. This method allows us to identify incorrect effects and, in some cases, to identify preconditions that may have caused incorrect values in the effects and investigate this further. However, identifying these effects can only be done in certain situations where there is a predefined pattern in the preconditions that explains how they affect the effects. Currently, this is only true for numeric arguments, where a calculation is done as part of the effects. In other cases, it will not be possible to identify these problem preconditions. If we cannot find a problem precondition then we must assume that the fault lies with the relevant effect. This is thus removed during refinement. However, since we are aware that there may be another undiagnosable cause of this failure, we do not wish to remove the original rule, that could possibly have the correct effects, from the ontology entirely. This is thus kept in the ontology but is not used during planning. If we encounter a problem with this rule in future, we will be able to refer to the original rule to see if it would have performed better in this case. If so, it can be reinstated. We will then be aware that this rule will not perform correctly under all circumstances; however, if the preconditions are correct, the effects should be instantiated properly. This will also lead us to the conclusion that there is an undiagnosed problem in the database that was the genuine cause of the problem. However, we will be, at this stage, unable to diagnose what the problem is. We can only hope that it will not cause problems during planning, or that if it does, we will then be able to diagnose why.

The Shapiro algorithm is summarised in Figure 5.3. It traces back through the justification to determine where the value of the problem precondition was last changed. If this was a fact in the original ontology, it removes this fact. If this fact was an effect of a previous action, it checks with the agent that performed that action as to what it believes the value of the fact should be. If this agrees with the value queried by the service-providing agent at failure, then the effects of that precondition are altered. The Shapiro algorithm does not, at present, investigate what happens if these two service-providing agents disagree with each other, as we have made the assumption that service-providing agents do agree with each other. If we were to remove this assumption, the PA would need to make judgements about which of these agents was likely to be the most reliable. The Shapiro algorithm also does not investigate failure caused by complex interactions between the preconditions and the effects, such that an error in the effects is caused by an error in the preconditions, as described above.

5.4 Diagnosis Failure

In some situations, regardless of how sophisticated the diagnostic process is, it will be impossible to provide a precise diagnosis of the cause of failure. This is due sometimes to the assumptions we have made about the behaviour and helpfulness of other agents, and sometimes to problems inherent in finding extra information to add to an ontology in a fully automated way.

The first issue arises because we assume that agents are not willing, or not able, to reveal their entire ontology to another agent (see agent assumptions in Section 6.2). Thus an agent must surmise what the problem is through the interaction that has taken place, and through further specific questions that are put to the service-providing agent by the PA.

The second issue arises because there is not always an automated source for informa-

tion missing from an ontology, or because the source of this information is unlocatable. Consider the situation in which a human user adapts an ontology, which is then used by an agent. The user intends to deal only with dollars, and therefore does not consider currency information relevant, thus representing money as *(Money ?Amount ?Agent)*. However, over time the purposes of the agent deviates to some extent, or perhaps, if it is using an off-the-shelf ontology, the authors of that ontology decide that currency information would be useful, and it encounters an agent which represents money as *(Money ?Amount ?Agent ?Currency)*. The original agent can easily diagnose and perform propositional anti-abstraction on the predicate *Money*. However, how is it to instantiate this extra argument in all the instances of *Money*? The knowledge that all money within this agent's ontology is in dollars exists only in the mind of the original developer, and is impossible for the agent to access. The simplest approach to this problem is to have a default value indicated for each set of subclasses; for example, the subclasses of currency might have a default value of dollars, as illustrated in Figure 5.8: the double line indicates the default value.

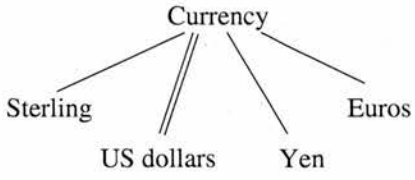

Figure 5.8: Subclasses of Currency

Thus if we have an uninstantiated value of class *currency* that needs to be instantiated, and there is no specific information about how this should be done, we would choose *dollars*. If no default value were given, the choice would be made arbitrarily. This method is preferable to randomly choosing a value, but it nevertheless introduces some amount of uncertainty, as choosing the default value is merely an educated guess, rather than a certainty.

Another approach would be to have certain agents in the system that are deemed to be knowledgeable on such matters. For example, it may be possible to put a question

(Money Me 1000 ?Currency) to a *bank agent* and expect that agent to be able instantiate *?Currency* appropriately.

5.4.1 Situations that Lead to Diagnosis Failure

There are two reasons why precise diagnosis is impossible:

1. The diagnosis system does not have sufficiently many or sufficiently sophisticated diagnostic algorithms.

   This effect can be minimised by thorough analysis of what kind of communication problems are likely to occur, but it is hard to eliminate this altogether.

2. We cannot extract sufficient information to fully refine the ontology, either because our assumptions about agent behaviour mean that they will not provide this information, or because this information does not exist in an extractable form (see the dollars assumption described above).

   We have briefly explained our assumptions about how much information we expect other agents to provide in Section 5.3.1. These are explained in more detail, together with the reason we feel these assumptions are important, in Section 6.2.

   The latter problem, of the information not existing in an extractable form, is also impossible to eliminate, since we do not wish to put restrictions on the developers of agents, but rather deal with the ontologies that are naturally encountered. It may be possible to build in some intelligent guesswork feature to the system: for example, if the agent is associated with a US user, and the currency of the US is known to be dollars, then, in the absence of any explicit information, that agent can guess that dollars is the appropriate instantiation. We can also allow the user to build in educated guesswork through the use of default values. However, there will be many situations in which guessing is not possible or feasible.

In these situations, there is nothing that can be done, unless there is some kind of oracle agent, such as the *bank agent* discussed above, or perhaps an oracle agent that is able to link, for example, location to currency, so that it might be able to inform an agent associated with a US users that its currency is likely to be dollars.

5.5 Implementing Refinements

Once the diagnostic process is complete, implementing signature refinements is not theoretically difficult. Through the diagnosis process we know exactly what we need to add to the ontology and what we need to remove. Implementing this is really just a question of text manipulation. However, there remain two difficult issues behind implementing refinements:

- **Determining the scope of refinement**

  Consider, for example, the situation in which a precondition *(Money PA 100)* is refined to a precondition *(Dollars PA 100)*, where both *(Money ?Agent ?Amount)* and *(Dollars ?Agent ?Amount)* are predicates defined in the ontology. In the particular rule where this predicate is being used, it is clear that the wrong predicate is being used. We must therefore remove the precondition *(Money PA 100)* from this rule and replace it with the precondition *(Dollars PA 100)*. However, it is not clear whether any further steps should be taken. It may be the case that it is never correct to refer to *Money*, and that every occurrence should be changed to some specific currency. However, not only would this create enormous difficulty, for how could we tell which currencies should be used, it is also unjustified. Why should we conclude that, since this was the precondition required for this rule, it must be required in all situations? The approach we take to this is to err on

the side of caution. Perhaps the correct thing to do would be to refine every instance of this predicate; however, because we cannot be sure of that, we refine only this specific instance, and refine others, if necessary, when they cause problems. It would be possible to build heuristics into the diagnostic process that would change all occurrences of a predicate if single instances had already been changed many times. However many occurrences we had encountered, we could never be certain that this was the correct thing to do, but it may be considered to be the best guess, given that it is always possible to refine the altered predicates back to the original if necessary.

Even more problematic is domain anti-abstraction. In predicate refinement, as discussed above, we can at least be certain that the specific problem is incorrect: we say with certainty that *(Dollars ?Agent ?Amount)* should have been used in this particular rule in place of *(Money ?Agent ?Amount)*. However, with domain refinement, we are dealing with particular instantiations of arguments. If we have a precondition *(Location ?Agent ?Place), (Class ?Place Place)*, which is instantiated in this particular instance as *(Location PA Edinburgh)* and this clashed with a surprising question from the other agent: *(Location PA Edinburgh_High_Street)*, this may indicate the need for domain anti-abstraction, because the class of *Edinburgh_High_Street* is *Street*, which is a subclass of *Place*. However, the agent's question does not conflict with our ontology: *Edinburgh_High_Street* is inevitably also of class *Place*, and so fits in with the class restrictions of our ontology. Thus we cannot be sure, even in this particular instance, that we need to refine this predicate. Since this is the case, we also err on the side of caution to avoid creating rules that cannot be fulfilled; see Section 7.3.2.

If the agent's question were *(Location PA Edinburgh_Castle)*, then we would be able to identify a mismatch: if I expect an individual of class *Street* and am asked about an individual of class *Building*, there must be some mismatch in

the class information. In this situation, one could either assume that the correct class for the argument is *Building*, or one could assume that our ontology is over-constrained, and that the correct class of the argument is *Place*. The dilemma in such situations is whether to diagnose domain refinement, or to simply assume that an incorrect class has been used. This would be easier to resolve if we had historical information, as discussed in Section 8.4.1, as this would tell us whether this particular problem had arisen previously. In the absence of that information, there is nothing to lead us to suspect it is more likely that domain anti-abstraction is applicable than that we have simply got an incorrect class. We therefore prefer the latter assumption, as this leads us to an over-constrained ontology, where we may have to do more work to achieve a goal than necessary, rather than an under-constrained ontology, where actions that we believe to be executable lead to failure.

This issue is not a problem for domain abstraction, because, since we are dealing with superclasses rather than subclasses, there will be a clash. That is, if we have a class restriction *(Class ?Place Street)*, and we encounter a predicate instantiated *(Location PA Edinburgh)*, this is clearly not compatible. A copy of the original instantiation could be retained and flagged as unusable, because this may contain important information concerning the uninstantiated argument. If, for example, the old class was *place* and an instantiation was *europe*, this would be incompatible if the class was altered to *country*. However, this would reveal important information about the way in which the new argument should be instantiated: we cannot tell exactly what it should be, but it is sensible to assume it must be a European country. The system is not currently able to perform any reasoning with such information; this is a task for future work, see Section 8.

- **Implementing theory refinements**

  The problems inherent in implementing theory refinements have already been discussed above to some extent. Sometimes signature refinement will be a matter

of refining a single signature object, as for domain and predicate anti-abstraction. However, sometimes signature refinement entails theory refinement. Most noticeably, this happens with propositional anti-abstraction. As has previously been discussed, this is both extremely hard (or sometimes impossible) to implement and also outside the main scope of the project. Therefore, we do not wish to put too much effort into solving this problem. However, we do need to decide what our approach is going to be, since the system cannot perform appropriately if this is not solved to some extent. Since the system is designed to keep replanning and attempting to execute plans after refinement has occurred, until the goal is eventually reached, it is essential to keep the ontology consistent after refinement, otherwise this process will be impossible. This means that all instantiations of signature objects (which are themselves theory objects) must be consistent with the definition of that signature object. Thus if a unary predicate is refined to a binary predicate, for example, then all instantiations of this predicate must also be made binary.

We deal with the problem of theory refinement after propositional anti-abstraction through the use of meta-variables. We insert a meta-variable in the correct place in every occurrence of the predicate. The class of this predicate is restricted through the predicate definition, which will have been refined to include the class of this extra predicate. The presence of these meta-variables means that the ontology remains consistent, but causes some problems during replanning. In order to produce an executable plan, it is necessary to fully instantiate all the actions in the plan. Furthermore, during the planning process, it will often be impossible to form a plan if preconditions of actions are not fully instantiated. If propositional refinement is performed, there is always exactly one occurrence of the predicate that is fully instantiated: the occurrence that led to the diagnosis of propositional refinement. Due to the surprising question, we have information about how this extra argument should be instantiated. However, we have

no information about how to instantiate any of the other predicates. The meta-variable will be instantiated to the default value or, if none is given, the first individual that is of the correct class. This is not a perfect solution, as it allows the possibility of error creeping in: we cannot be sure we are choosing the correct instantiation, even if a default value is given. However, this approach allows us a solution that is reasonably likely to succeed, and which enables us to maintain a consistent ontology and thus continue planning. Dealing more fully with this problem is outside the scope of the project. It seems that there is much that can be done to improve this situation, but nothing that could be done to remove the problem entirely and allow us to instantiate all occurrences with certainty.

5.5.1 Translating the Refinements

Once the diagnosis has been made, the refinement information has to be translated into KIF, so that it can be used to alter the KIF ontology correctly. The refinement translation process returns a list of items that are relevant to the refinement. The contents of this list depend on the type of refinement. As an example, we list below the four anti-abstractions, with an example of the predicate that calls the refinement system, and an explanation of what this information is.

- **Propositional anti-abstraction**: an extra argument is added to a predicate:
  Refinement command:
  refine(propositionalAA,['Money','Currency',3,2]).
  The information in this predicate refers to, in order:
  - The predicate name, suitably translated;
  - The class of the new argument, appropriately translated. If this class is not already contained in the ontology, this will entail another kind of refinement: an alteration of the class hierarchy will be required. This is

determined during the diagnostic process, and an *add class* refinement performed in addition to this;

- The position in which the new argument must be placed;
- The arity of the original predicate

- **Domain anti-abstraction**: the class of one of the arguments is changed to a subclass:

  Refinement command:

  refine(domainAA,['Money','Currency','CurrencyType',2]).

  - The predicate name, suitably translated.
  - The original class of the argument, suitably translated.
  - The new class of the argument, suitably translated.
  - The position of the argument that is to be altered.

- **Precondition anti-abstraction**: a new precondition is to be added to a rule:

  Refinement command:

  refine(preconditionAA,['Buy','(Money ?Agent ?Amount)']).

  - The name of the rule that is to be altered, suitably translated.
  - The new precondition, suitably translated. This means that not only are all the names in the precondition altered appropriately, but also the syntax: for example,

    money(Agent,Amount) becomes *(Money ?Agent ?Amount)*.

- **Predicate anti-abstraction**: the name of a predicate is changed to a subclass:

  Refinement command:

  refine(predicateAA,['Money','Dollars','Buy-Rule']).

  - The old name of the predicate, suitably translated.
  - The new name of the predicate, suitably translated.

- The name of the rule in which this change should take place, suitably translated (see Section 5.5 for a discussion as to why this change is only implemented in a single rule).

The above list mentions the *refine/2* predicate, which is the predicate that calls the refinement process. See Appendix B for further information about how this refinement predicate works, and for samples of the coding.

5.6 Summary

This chapter describes the central functionality of ORS: diagnosis and refinement. It outlines what the various components of the system are and how they interact. It explains what kind of ontological mismatches we are expecting to encounter, and how we propose to diagnose and refine these mismatches. Figure 5.9 illustrates the proportion of ontological changes the system can handle. All of the ontological objects are listed, with the types of changes that could be made to them. These are divided into the different ways in which these changes can occur. White boxes indicate situations in which accurate diagnosis and refinement can always be made. Blue boxes indicate situations in which accurate diagnosis and refinement can sometimes be made: sometimes a plausible but uncertain diagnosis and/or refinement can be made; sometimes no diagnosis or refinement is possible. Red boxes indicate situations in which accurate or even plausible diagnosis and refinement cannot be made.

Adding new ontological objects is difficult because this can require a great deal of information. For a class this is not a problem, because we have a very simple definition of what a class is. If an unknown class is mentioned by another agent, we can be sure it is possible to determine its name and superclass. Deriving all the necessary information for predicates is sometimes possible; for action-rules, it never is. Dealing with action-rules is difficult because very little information about these is revealed through agent

communication. Some of these shaded boxes are not certain to succeed, but work very often, such as adding missing preconditions; some are difficult to diagnose and refine correctly, such as changes in effects: information about this is only extractable through the Shapiro algorithm. Sometimes, our approach is not good for detecting ontological mismatch simply because it doesn't apply in the planning context in which our system is set. For example, it is hard to diagnose that a rule has an extra precondition, because this will never lead to plan failure, although investigating questions that are not asked when they are expected could be helpful here.

Many of the ontological mismatches that we are not capable of diagnosing are those that are of a more random or chaotic nature, and thus it is very difficult to develop any kind of diagnostic techniques that would be able not only to identify the problem but to correctly diagnose a solution. Although it is of some use to diagnose problems even if we cannot find a solution for them, it is clearly more useful to concentrate on situations in which both diagnosis and refinement are possible, and this has been our approach. There are some cases, outlined above, where, in looking for fixable problems, we may encounter problems that we do not know how to repair. In these cases we can only flag these ontological objects as unusable without adding any information about how they may be made usable. This information is certainly of some use in making executable plans, but we concentrate on situations where we can not only diagnose but also fix problems.

As explained in the previous chapter, the techniques we have outlined are only appropriate for first-order ontologies. The same kind of approach would be applicable for non-first-order ontologies, but the potential mismatches we would identify would be different. Additionally, if our techniques were applied to a first-order ontology with more complex or additional ontological objects, there may be areas in which our techniques would not work; further research would be necessary here. In particular, our notion of an ontology has a rather simple approach to classes, which are used merely

to create the class hierarchy and to designate the classes of individuals. It is common in ontologies to use a much more sophisticated approach to classes, where much additional information is assigned to classes. The same approach can be used to develop techniques to deal with these, but currently the ORS can only deal with simple classes.

<table>
<tr><td>Predicates:
Change of name</td><td>Change to subclass:
Predicate anti–abstraction</td><td>Change to superclass:
Predicate abstraction</td><td>Change to random name</td></tr>
<tr><td>Change of arity</td><td colspan="2">Add an argument:
Propositional anti–abstraction</td><td colspan="2">Remove an argument:
Propositional abstraction</td></tr>
<tr><td>Change of argument class</td><td>Change to subclass:
Domain anti–abstraction</td><td>Change to superclass:
Domain abstraction</td><td>Change to random class</td><td>Switch arguments</td></tr>
<tr><td></td><td colspan="4">Add a new predicate</td></tr>
<tr><td>Action–rules:
Change in the preconditions</td><td>Add a precondition:
precondition anti–abstraction</td><td>Remove a precondition:
precondition anti–abstraction</td><td colspan="2">Change an existing precondition</td></tr>
<tr><td>Change in the effects</td><td>Add an effect</td><td>Remove an effect</td><td colspan="2">Change an effect</td></tr>
<tr><td></td><td colspan="4">Add a new rule</td></tr>
<tr><td rowspan="2">Classes:</td><td colspan="4">Change of superclass</td></tr>
<tr><td colspan="4">Add a new class</td></tr>
</table>

Figure 5.9: Refinements Covered by the System

Chapter 6

Subsystems of the ORS

6.1 Introduction

In this section we introduce the subsystems on which the diagnosis and refinement are built.

6.2 Agent Communication System

The entire process of diagnosis and refinement is being carried out within the framework of an agent communication system; thus the development of such a system is a vital part of the ORS. At the same time, the agent communication system is merely part of the framework and is not itself a focus of our research.

Developing an agent communication system can be a problem of significant complexity. There are many issues that one might wish to consider concerning agent behaviour, agent languages, agent protocols and such like. For our purposes, however, all of these concerns are side issues. We require an agent communication system as a platform for

exploring our ideas of diagnosis and refinement and thus do not wish to address these these agent issues in much detail. What we require from the agent communication system is a platform to allow agents to pass messages to one another. All we require is that the agents can freely communicate, can ask for actions to be performed and can ask questions concerning these actions if necessary. We make whatever assumptions are necessary concerning the interaction of agents. We feel free to make simplifying assumptions about the agents in all areas other than those we are directly investigating. Specifically, we assume that agents are honest, helpful, are always available and can be found in the places where we expect them to be. We assume that they will perform tasks that they are capable of performing if the necessary conditions are fulfilled, and hence refusal to perform a task means that either they are the wrong agent, or that they believe that the preconditions are not fulfilled. We assume that communication is always reliable and that messages are never simply lost.

However, we do not wish to make too many assumptions about the agents' behaviour regarding their ontologies. The basis of our research is the ability of agents to interpret the ontologies of other agents in light of their actions and to analyse how these other ontologies differ from their own. If we assume too much about how willing or capable the other agents may be to divulge information about their ontologies, then our research will be limited in applicability. It is very difficult to make predications about how we can expect agents to behave, since there is great disparity between different agents and a high degree of freedom for the designer of agents to make his own mind up about these things. Therefore, in order to ensure a reasonable generality to our system, we are making the assumption that agents will not reveal very much about their ontologies.

We assume that agents respond only to the following types of communications:

- **request**$(a\vec{x})$, where a is an action name, and $\vec{x}$ are the appropriate arguments.

  This represents another agent asking for action a to be performed. The agent receiving this query will reply that the action has been performed if it is able to

perform the action, and if it can ascertain that all the preconditions of the action are fulfilled (possibly through communication with other agents), and will reply that the action has not been performed otherwise.

- **query**$(f\,\vec{x})$, where f is a predicate name, and $\vec{x}$ are arguments.

  This represents another agent asking for information concerning a specific predicate. There are two possible situations:

  - All of the arguments are instantiated. In this case, the agent receiving the query will reply as to whether it believes this instantiated predicate (or fact) to be true or false. The agent may find this predicate (or its negated version) as a fact in its ontology, either because it originally believed it to be true or because it has been made true during previous interactions, or it may be able to infer the truth of this fact.
  - One or more of the arguments are uninstantiated. In this case, if the agent has a fact that is a fully instantiated version of this predicate, it will return this, or, if there is more than one fact that will match, it returns one of these. For example:
    query: (*Money PA ?Currency ?Amount*)
    reply: (*Money PA Dollars* 1000),
    or
    query: (*Money PA Euros ?Amount*)
    reply: (*Money PA Euros* 500)
    If the agent has no fact in its ontology that matches this predicate, it will reply that this is false.

There is no facility for forcing an agent to reveal more information than this. In particular, there is no way of directly enquiring about action rules.

We believe that this approach is the most plausible one to take. The kinds of interaction

described above are standard forms of agent interaction, and it is reasonable to expect most agents encountered in a Semantic Web like environment to be able to behave in such ways. The kind of interaction that would be required to discover more about another agent's ontology is not part of standard agent interaction, and we do not believe that it is reasonable to expect agents that have not been designed specifically for this purpose to be able to behave in such ways. It is important for the applicability of the system that we expect agents who make use of it only to behave in ways that are standardly exhibited by agents, rather than to expect them exhibit specialist behaviour. Additionally, we cannot assume that agents would be prepared to reveal more than this about their ontologies because of the security issues this would entail. Many agents have commercially sensitive information: for example, an agent that is providing a particular service for a fee would not wish to reveal very much information about how this was done, as this information is of material value to it (or its owner). It may also be the case that the underlying ontology of an agent is completely different to the format that is used for communication; it would thus be completely unhelpful to have direct access to the ontology. For these reasons, we assume that agents will reply to direct questions and requests for actions to be performed only. We use a purely message-passing architecture; agents can only interact by message exchange. There is no global clock, no direct observation of other agents' definitions, no shared knowledge.

Another important assumption we make is that the PA alone is concerned with bringing its ontology in line with that of other agents. There is no negotiation about which agent should alter its ontology, and the service-providing agents are not concerned that their ontologies are not in line with the PA. When the PA identifies ontological mismatches, it alters its ontology in order to eradicate these; it does not question to what extent it ought to believe the other agents. We believe that these assumptions are plausible in the scenario we are investigating, because the PA's interaction with other agents consists entirely of asking service-providing agents to provide services for it. If it wants these services to be provided, it has to conform to what the service-providing agents what

of it, and therefore is always prepared to take on the submissive role. Additionally, making this assumption simplifies the situation to the extent that an initial working system can be produced. However, making this process more sophisticated, by making the refinement process more interactive between the agents, would be an important addition to the system. This issue is discussed further in Section 8.4.2.

In summary, our assumptions about agent behaviour are:

- Agents have particular roles, which they stick to throughout the course of the interaction; for example, a *buying* agent cannot simultaneously be a *selling* agent;
- Agents are helpful and honest; what they tell us may not be true, but this occurs because they are mistaken in their beliefs, not because they are deliberately dishonest;
- We are aware of all the agents involved in the system, and the world only changes through these agents interacting the PA;
- Agents are able to communicate only by exchanging messages;
- Agents will always perform actions for one another if possible; an action will only fail if an inappropriate agent is contacted or if the preconditions of the action are not fulfilled;
- Agents share a common protocol; it is only in the content of the messages that mismatches occur.

As a result of these assumptions, the implementation of the agent communication system is relatively straightforward.

6.2.1 Implementation of the Agent Communication System

The Agent Communication System is based on Linda, which is a set of language extensions based on a tuple-space, where a tuple-space is a shared memory storage abstraction which provides a global storage mechanism for nodes operating within a network [Patterson et al., 1993]. The tuples, together with a small set of primitives, provide for interprocess creation, communication and synchronisation. We are using a version of Linda written in Sicstus Prolog [Sicstus, 2005]. One process runs as a server and one or more processes run as clients: thus the agents in the system are Linda clients, with a Linda server present to facilitate agent communication. The server acts as a blackboard which agents can write to, read from and delete messages from. This is similar to a broadcast mode of communication, except that reading a message consumes it, so that it is no longer available.

The agent communication system that we build on top of this is fairly basic. The agents send queries to one another by using the writing predicate *out/1*, where the argument taken by this predicate is the outgoing message, specified by the agent protocol described in 6.2.3. Agents read messages for themselves and, once read, remove them by using the read-and-delete predicate *in/1*. There are two versions of this predicate: the standard *in/1* and also *in_noblock/1*. When the former is used, this causes the agent concerned to wait until an appropriate message appears on the blackboard, only continuing with other tasks once this has occurred. When the latter is used, the agent concerned checks the blackboard for a message, and fails if an appropriate message is not present. Service-providing agents have a more limited function than the PA; they are there simply to perform tasks for other agents, and will wait around until such a task is required from them. Thus they use the *in/1* predicate. PAs, on the other hand, are performing other tasks than simply communicating with other agents: for example, forming plans; and hence use the *in_noblock/1* predicate. In a more complex system, such as the one we envisage developing as a future extension to the system,

service-providing agents may also be PAs; they may perform many actions including plan execution in addition to performing their given task. Thus they would then use the *in_noblock/1* predicate. The disadvantage of using this predicate is that it creates the possibility of agents missing messages, and failing because a message was not present, when in fact the message may have appeared after a time lapse. In our system, we safeguard against this happening by calling the *in_noblock/1* command and, if it fails, sleeping for two seconds and then calling it again. If it fails this second time, it is assumed that the communication has failed for some reason. This simple safeguard is sufficient in our system where the interaction is not complex, because most of the agents are doing nothing but waiting to perform tasks. In a more complex system, such concerns would become more difficult to control, and a more sophisticated approach may be necessary.

6.2.2 Algorithm of Service-Providing Agents

The algorithms of the PA are described in detail in Chapter 5. Here, we outline the algorithm of service-providing agents, illustrated in Figures 6.1 and 6.2. This is much simpler than the algorithms of the PA, since the functionality is much less. The service-providing agent must check whether it can, in fact, perform this action and, if so, must step through the preconditions one by one to verify if they are correct. Preconditions need to be dealt with differently: the truth of some is established through asking the PA; the truth of others through asking other agents (for example, an airline agent may be able to provide information as to whether there is a free seat on a given flight); the service-providing agent may be able to ascertain the truth of other preconditions itself; and others may be ascertainable by the service-providing agent once certain information has been received (for example, a precondition that states the price of a seat must be under a certain value can be ascertained once the airline agent has responded as to what price seats are still available.) If any preconditions do not hold, failure is

reported. If all of the preconditions do hold, the service-providing agent updates its ontology according to the effects of the action, and reports success.

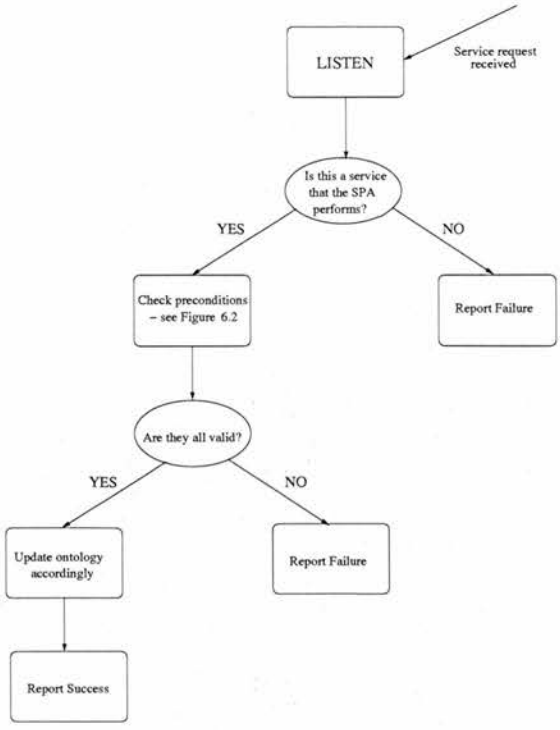

Figure 6.1: Service-Providing Agent Algorithm

6.2.3 Agent Protocol

We have developed a simple agent protocol for use during agent communication. The protocol for performing actions is illustrated in Figure 6.3.

When a request to perform an action is made of a service-providing agent, that agent can respond in three ways: report failure in performing the action (*problem*), successfully perform the action (*ok*), or request further information from the request-making agent (*query(X)*). If a query is made, the PA must respond to it, though this response may indicate an inability to correctly answer the query. Again, the service-providing agent can respond in any of the above three ways. This process is repeated until the

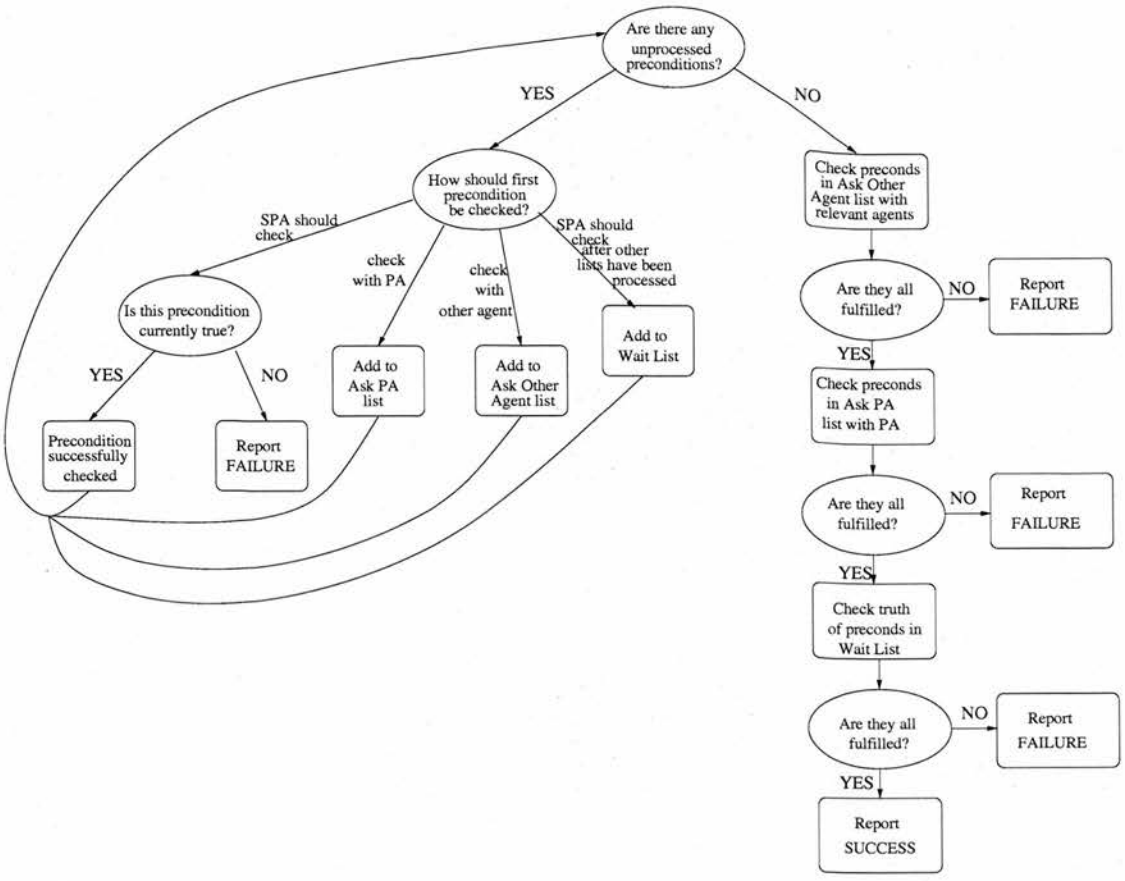

Figure 6.2: Precondition Checking Algorithm

service-providing agent is satisfied that all the preconditions for the action are fulfilled. This means that the number of queries made by the service-providing agent cannot be greater than the number of preconditions for the action (it is usually fewer, because some of the preconditions can be checked without making queries). Once all the preconditions have been checked, the service-providing agent has only two options of response: *ok* or *problem*. The protocol terminates whenever a response of *ok* or *problem* is received.

This protocol contains only two performatives: query and reply (equivalent to ask and tell). These are of the form:

query(SendingAgent,ReceivingAgent,QueryType,Query)

reply(SendingAgent,ReceivingAgent,QueryType,Answer)

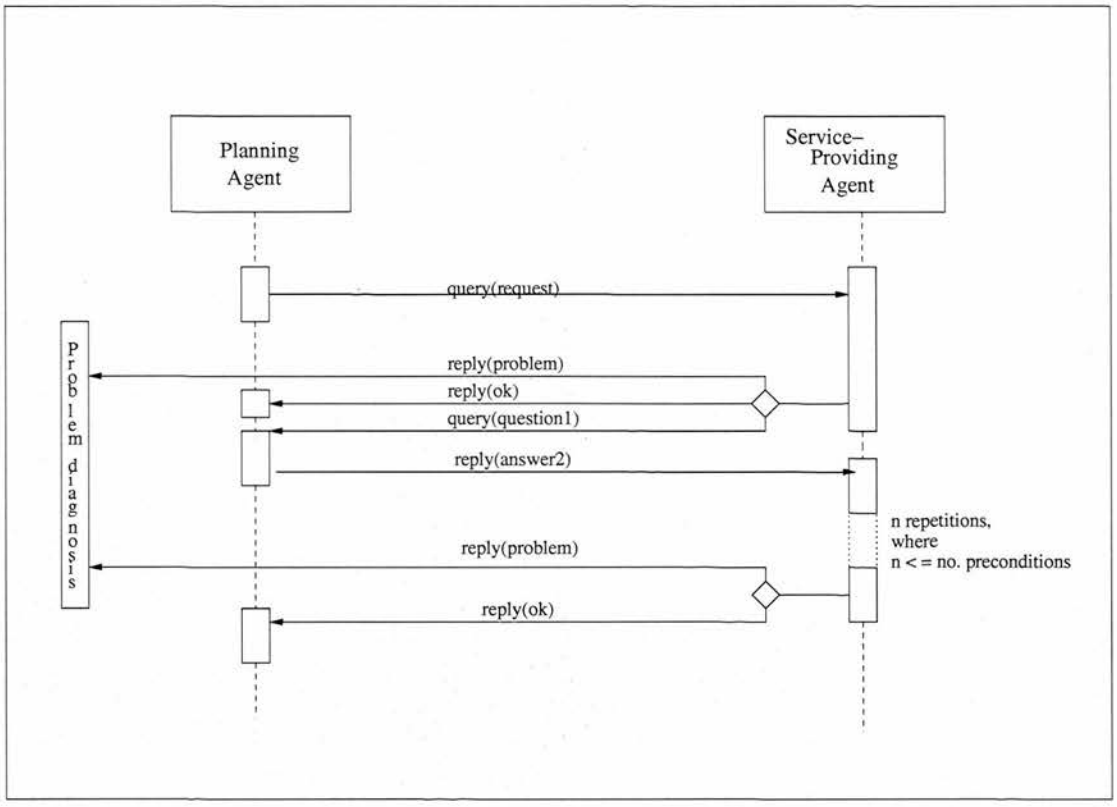

Figure 6.3: Action Performing Protocol

Thus the agents are all constrained to either asking questions (which can include requesting actions to be performed) or responding to these questions, which may be with an answer to the question, or with an intermediate question if the answer cannot be immediately ascertained.

These two performatives are subdivided into different types, determined by the third argument, *QueryType*. This argument can take the following values:

- *Request*: this is a request for the Receiving Agent (service-providing agent) to perform a task for the Sending Agent (PA). The ultimate response is *ok* if the task has been performed, or *problem* if it has not. There may be intermediate queries and responses whilst the receiving agent is tries to ascertain whether or not it can perform the task.

- *Question*: this is a request for the Receiving Agent to respond to a direct question from the agent. The response to this may be *yes* or *no*, or it may be an instantiation of the query (if the query contained uninstantiated variables).

Since we require only two performatives, we have not made use of standard performative languages such as FIPA-ACL [FIPA-ACL, 2002] or KQML [KQML, 2005]. The kinds of functionality that these languages are designed to facilitate are similar to, though more complex, than the functionality that can be performed in our system. The purposes of performatives in FIPA-ACL can be grouped into five categories: *passing information*, *requesting information*, *negotiation*, *performing actions* and *error handling* [Wooldridge, 2002]. Our system is designed to deal with four of these categories; negotiation is not dealt with. However, the richness of the FIPA-ACL specification allows these to be dealt with in a much more sophisticated manner than is possible in our system. For example, there are five different performatives which are designed for passing information: $confirm$, $disconfirm$, $inform$, $inform\text{–}if$ and $inform\text{–}ref$. In our basic performative language, passing information is performed with only one performative: *reply*.

Since it was important to keep agent communication simple, we decided to avoid making use of the richness of a performative language such as FIPA-ACL or KQML. However, many of the extensions we would like to make to the system, discussed in Chapter 8, require more complex agent communication. In order to allow for this, it would be advantageous to make the system FIPA compliant; thus we can take advantage of the additional functionality provided by such a language. Additionally, making the system FIPA compliant would make it accessible to a much broader community, as FIPA-ACL is the standard performative language. Without a shared performative language, successful interaction is generally impossible (though some work is being done on adding some flexibility to this [McGinnis and Robertson, 2004a, McGinnis and Robertson, 2004b]). Thus making the system FIPA-ACL compliant would greatly increase the ap-

plicability of the system. These issues are discussed in Section 8.2. Another approach we have investigated is conducting agent communication with agent protocols, such as the electronic institution framework [Esteva et al., 2001, Sierra et al., 1998]. Our ideas about basing our refinement techniques in such a system are discussed in Section 8.4.6.

6.3 Planning System

The role of the planning system is to return a plan for achieving the desired goal, which is annotated with an appropriate justification. The justification is required in order to provide information about how the plan relates to the ontology: which rule was used to perform a particular action; what the preconditions of this rule are; why we believed each of the preconditions to be true in the given situation; what the effects of the rule are.

As discussed in Section 5.3.4, this justification is not always necessary. Sometimes, the information revealed directly through the agent locutions is sufficient to link the cause of plan failure to a specific part of the underlying ontology. However, in some situations this information is not sufficient; in these cases, we need to make use of the plan justification through the Shapiro algorithm.

If we wish to extract this information from a planner, we require a planner which will use such first-order reasoning techniques to build a plan, and then be willing and able to divulge this information to the user. A first-order planner - a Prolog planner, for example - would be able to do this and would not be difficult to build from scratch. However, such planners are seldom used in planning because their search techniques are inherently inefficient. They are usually very resource intensive, and it can often be impossible to develop plans that are more than a few steps long. It seems, therefore, very unwise to build a system that is dependent on such a planner. Even if we are

initially only using small plans that could be built on such a planner, we do not wish to build this limitation into the system.

On the other hand, efficient planners are not first-order and are not capable of returning the kind of information we need. A common search tactic is to take an ontology that appears to be first-order, together with all possible variables, instantiate the ontology in all possible ways and then perform a search through the propositional space. It is clear that such a search technique is not helpful for returning the higher-level reasoning information that we require. In addition, many planners are black-box in nature and unable to return much information about the planning process.

We have therefore developed a plan deconstructor, which will take a completed plan, provided by any sort of planner, and reconstruct a first-order justification for the plan. We are thus able to make use of an efficient planner and also to receive the reasoning information we require. This is discussed in detail in Section 6.3.2

6.3.1 Planner

Given that we have developed the plan deconstructor separately, the requirements of the planner were few. We simply require an efficient modern planner that is capable of producing linear plans, since these are the only kinds of plans the system currently deals with. Due to the modular nature of the planning system, the choice of planner is by no means fixed; we can slot in a different planner if we wish with minimal impact on the system. This would affect the interpretation of the plan; every planner represents plans in a slightly different format, and therefore the part of the system that reads and interprets the plan so that it can be read by the agent and the plan deconstructor would have to written independently for each planner. However, this is a very small part of the overall system; the rest of the system would not be affected.

6.3.1.1 Choice of Planner

There were two primary concerns for the choice of planner:

- it is preferably to have a modern, widely recognised planner;
- the choice of the specific planner should limit the system as little as possible.

The first concern is important for two reasons. Firstly, we wish the system to be as efficient as possible, and automated planning can be a timely and difficult process. Secondly, in order to make this work as widely applicable as possible, and in particular, to demonstrate to the planning community that this work makes a viable and important contribution to multi-agent planning, we wish to choose a planner that was already well-established and well-regarded.

The second concern entails avoiding making the entire system dependent on a planner that may at some time become outdated, but rather that it would always be capable of making use of the most advanced planners. With this in mind, it seemed desirable to use a planner that accepted knowledge in the PDDL representation. This choice does not limit the system to a great extent. If we wished to change to a planner that used another representation, the PDDL section of the translation process would have to be rewritten. However, the other aspects of the system would remain unaffected.

With the above concerns in mind, we chose Metric-FF as our planner [Hoffmann, 2003]. Metric-FF was developed by Jörg Hoffman as an extension to his planner FF. FF (Fast-Forward) [Hoffmann, 2005] is a domain independent planning system. It was awarded for outstanding performance at the 2nd International Planning Competition [PlanComp2, 2000], and was the top performer in the STRIPS track at the 3rd International Planning Competition [PlanComp3, 2002]. Metric-FF is designed to work

with PDDL-2.1 which incorporates, among other things, the ability to deal with durative actions [Fox and Long, 2003]. Metric-FF was the top performer in the numeric track at the 3rd International Planning Competition [PlanComp3, 2002].

6.3.1.2 Limitations of the Planner

Automated planning is very difficult; hence, although we have chosen a state-of-the-art planner, this does not mean that planner is without limitations. If the PDDL input is of sufficient complexity, Metric-FF may be unable to find a plan even if one exists. In such cases, the planner will run out of memory and not return any information as to whether a plan is possible. In particular, increasing the arity of a predicate greatly increases the complexity of finding a plan. We have found the presence of predicates with an arity of five or more creates problems for Metric-FF, and it is either extremely slow to produce a plan, or runs out of memory. Even if the class restrictions of the extra predicate are such that the complexity of the search space is not hugely increased: for example, restricting the class such that only two individuals exist that are relevant, hence only doubling the search space, Metric-FF may nevertheless fail to find a plan. This is because much of the work is done in pre-processing, where the class of the extra argument may not even be noticed.

The only refinement that is really affected by this limitation is propositional refinement. In order to avoid situations where the planner fails, we have restricted our examples of this refinement to predicates that have an arity no greater than 3, and thus, after refinement, have an arity no greater than 4. This is regrettable, and may create situations where it is impossible to diagnose the correct refinement. However, this is a limitation of modern planners, which comes about because automated planning is a very difficult task, and thus, if there is to be a planning element to our system, we cannot help but be restricted by this. Also, it is unusual to encounter predicates that have an arity greater than 4; most ontologies do not contain predicates of such high complexity.

6.3.2 Plan Deconstructor

The role of the plan deconstructor is to produce a first-order ontological justification for the plan produced by the planner. It is not attempting to reproduce the steps taken by the planner since this, as has been discussed above, will not provide the necessary information.

The justification for the plan is a list of justifications for each action in the plan, and is of the form:

`[lastJustification,...,firstJustification].`

Each action justification is of the form:

`[Action,RuleNo,Preconds,Effects,State,Situation]`

and contains the following information:

- the name of the action
- the identifier for the rule
- the instantiated preconditions for the action
- the effects for the action - these will generally be instantiated in the justification, but in some cases some arguments may depend on information provided by the agent performing the action for us; these must necessarily be instantiated after the justification is produced, during execution.
- a version of the *state*, updated for that action. The state contains information about how fluents are changing during plan execution, and is explained in more detail below.
- a version of the *situation*, updated for that action. The situation is a list of all the plan steps that have been executed so far, thus describing the situation in which the preconditions of each action are held to be true. The situation is updated by appending the current action to the situation list.

The state is perhaps the most important part of the justification, and it is to this that we turn when we require information about what may have caused errors earlier in the plan execution. The state is of the following form:

[[[values of facts altered in sit1],[sit1]],
[[values of facts altered in sit2],[sit2]],...]

The values in the state are very similar to the effects for each action, but are not identical. The state values represent processed versions of the effects: for example, an effect may be a calculation to be performed, e.g., *NewAmount* = *OldAmount* - *Cost*. When the state is updated, this calculation will be performed, and any effect that contains the variable *NewAmount* can be instantiated, but the calculation itself is not listed in the state. We list both the effects and the state in the justification for convenience, since we at times need to refer to one and at other times to the other. It would not be especially complicated in such cases to calculate the state from the effects, though not vice versa. However, it is more useful to carry this information in a list created during plan deconstruction, so that it can be referred to at any point without further calculation.

6.3.2.1 Implementation of the Plan Deconstructor

The plan deconstructor follows the specification described in Section 6.3.2. The deconstructor, written in Prolog, uses the Prolog version of the ontology. It performs the following actions:

- it builds a list of facts of the initial state by observing which facts are true in the original ontology;
- it attempts to meta-interpret the plan by finding an action with the correct name in the ontology, and then verifying that all the preconditions of that action hold in the current state;

- it builds up a new state by creating a list of all the effects of the action that has just been performed, and appending them to the situation, which describes which actions have taken place thus far. This list contains, in STRIPS terminology, both the add-list and the delete-list, as it contains information about what has been made true and what is no longer true. This new list is appended to the current state;

- these lists of effects of the actions that have been performed are appended to the relevant situations in the justification of the plan, and formed into a list containing all the effects of actions with the relevant situation: this list explains why each fact is considered to be true, and at what point it came to be made true.

We give below an example of what the deconstruction might look like after the first action has been meta-interpreted:

Action name:

[[(ConverPaper IsabellePaperPdf Lucas IsabellePaperDvi),

rule identifier:

rule6,

Preconditions:

[(Class IsabellePaperDvi DviPaper),(Class Lucas Agent),(HasPaper Lucas IsabellePaperDvi)],

Effects:

[(Class IsabellePaperPs PsPaper),(Class Lucas Agent),(HasPaper Lucas IsabellePaperPs)],

State:

[

[[(Class IsabellePaperPs PsPaper),

(HasPaper Lucas IsabellePaperPs),(Class Lucas Agent),

[(ConvertPaper IsabellePaperPs Lucas IsabellePaperDvi),start]],

[[(AccommodationInfo Cade 50),(HasPaper Lucas IsabellePaperDvi),

(Location Cade Miami),(Location Lucas Edinburgh),(Money Lucas 1000),

(RegistrationFee Cade 200),(Flight Edinburgh Miami 300)],

[Start]]

],

Situation:

[(ConvertPaper IsabellePaperPs Lucas IsabellePaperDvi),Start]]]

The first four items give information concerning the most recent (and, in this case, the only) action: here, it is the action *ConvertPaper*. The fifth item, the state, lists not only the information for this particular action, but gives a snapshot of the truth of each fact at each point in the deconstruction thus far. Thus the first part of it lists the facts that have become true after the first action has been performed; at this point, the situation is *[(ConvertPaper IsabellePaperPs Lucas IsabellePaperDvi),Start]]*, and this is part of the list of facts, so that the context in which these facts are true is immediately clear. The second part of the state lists the facts that are true at the previous step of the deconstruction: in this case, this is at the start. Again, that situation is attached to the facts. Finally, the situation (which, as discussed above, also appears in the state) is explicitly stated.

The plan deconstructor behaves in a similar manner to a first-order planner, except that the huge search problems encountered by first-order planners have already been dealt with by a more efficient planner. Thus the combination of the planner and the plan deconstructor combines the efficiency of a state-of-the-art planner, with the plan formation information provided by a first-order planner.

6.3.2.2 Relation of the Plan Deconstruction to Plan Formation

As has been discussed above, the justification provided by the plan deconstructor is an explanation of how the plan *might have been* formed, not how it *actually was* formed. There are two reasons why we choose to use this information, rather than information about how it actually was formed:

- For the kind of efficient planners we would want to use, it is usually impossible to retrieve such a justification.
- Even if it were possible to retrieve this information, it would not be the kind of first-order reasoning we require to investigate the problem, because these planners are propositional and use SAT solvers or similar to solve the search problems. Thus this information would not be very helpful for the kind of analysis we need to do.

It seems reasonable to assume that in many, or even most, cases, the justification provided by the plan deconstructor will point us to exactly the problem that is causing the plan execution failure. However, this may not always be the case; it may be that there is more than one way to justify a plan, and that the way that the plan deconstructor finds is not the one that will lead to the discovery of the problem that actually caused the failure. Certainly, any justification that is formed by the plan deconstructor will be a valid justification for the plan, and any valid justification for an inexecutable plan must be based on onto some kind of ontological error. We can be sure that there is some kind of error, highlighted by this particular justification, that would lead to the plan being inexecutable. Thus, if we diagnose and refine as normal, we will always reach a state where the ontology is closer than it was before, since one source of ontological mismatch has been patched, and possibly a state from which an executable plan can theoretically be developed (this will depend on whether the justification included

more than one ontological error). If we then replan, we would hope that the planner would use the refined ontology to produce a slightly different plan to the one produced previously, which would then be executable (or would fail at another stage due to another error). However, if there is more than one way to justify this plan, then it may be that exactly the same plan will be produced, since if there is still a way to justify the old plan, it is still possible that it could be developed from the updated ontology. Ideally, this will be a iterative process, where we go through the justifications one by one, each time refining the error on which that particular justification is based, until it is no longer possible for the planner to develop the inexecutable plan. This seems like a laborious process; however, we are, at each stage, learning more about the domain and refining the ontology in a way that may well be helpful to us later, and additionally, it seems unlikely that there will be many situations in which there is more than one differing justification for the same plan, and certainly not many situations where there are several.

6.3.2.3 Completeness of Plan Deconstructor

In order to be sure that this is a reasonable way to do things, we need to be sure that:

- If it is possible to develop a plan to reach a given goal from a given ontology, then the planner will be able to find such a plan.
- If it is possible to justify a developed plan from a given ontology (which, from a technical standpoint, is equivalent to it being possible to develop that plan from that ontology), then the plan deconstructor will find such a justification; *i.e.*, that the plan deconstructor is complete.

If the above two criteria are met, then we can be sure that deconstructing the plan separately to creating the plan will not lead to a situation where the ontology cannot

be refined appropriately. More specifically, it will not lead to the formation of a plan that cannot be deconstructed, nor will it lead to a failure to form a plan, where deconstruction of a potential plan, or formation of a plan using a first-order planner, is still possible.

Additionally, we can be sure that refining the plan by using a justification produced externally to the planner will always eventually end in the refinement of the precise problem in the original plan (barring limitations in the diagnosis and refinement processes). In most cases, unless we have many different actions with the same name, it is probable that plans can only be deconstructed in one way, and hence only one cycle of refinement is necessary until the plan is fixed for the particular point of plan failure. But if the two criteria are fulfilled, then we will always be able to find the correct refinement, even if this takes more than one refinement cycle. The maximum number of refinement cycles this could take is the number of ways in which the ontology differs from the ontology to which it is being mapped, which is equal to the number of different ways the planner may have formed the plan. It is usually not possible to get into loops. However, consider the situation in which PA is trying to execute the plan:

$P = A, B, C$; where A,B and C are actions.

Imagine there are two ways of justifying this plan:

A_1, B_1, C_1, or

A_2, B_2, C_2,

where A_1 and A_2 are rules that enable the performing of action A, and so on.

If plan P is not executable, there must be a flaw in both of the justifications: perhaps there are problems with the preconditions of rule B_1 and those of A_2. Imagine plan execution failure is encountered during the attempted execution of action C. If the first justification is being used, the problem will be traced back to the problem with B_1; this will be refined and replanning can commence. A justification for this inexecutable plan can still be found: the second justification, for example. However, such a justification

could not include rule B_1 unless that rule had an additional flaw that also led to plan execution failure. If B_1 is not able to be included in the justification, then we can always be sure that are getting closer to the situation where this inexecutable plan cannot be justified, and hence cannot be formed from the ontology. If B_1 had more than one problem, it is possible that problem could still be in a new justification. If the plan execution fails in a different place, and if this is traced back to the rule B_1, and if the refinement that was diagnosed to solve this problem was the exact inverse of the refinement that had previously been performed on B_1, to solve the previous cause of plan execution failure, then the process may loop. This scenario is clearly unlikely, but cannot be ignored. The simplest way to solve this problem is to check the record of past refinements to see if the same refinement has been performed more than once. If so, that ontological object should temporarily be excluded from use in a justification.

This process will stop as soon as the deconstructor finds the deconstruction that corresponds to the way in which the planner formed the plan. The presence of this upper limit of refinement cycles means that this will always be a finite procedure, unless we are not only using an infinite ontology, but also an ontology where there are an infinite number of rules with the same name, hence allowing for an infinite number of possible deconstructions of the same plan. A simple way to avoid all these problems is to allow each action name only one rule which can perform it, which is, in fact, standard in most ontological representations. Two rules would then be able to produce the same effects, but they would have to have different names. Then there would be only one way of justifying each plan.

6.3.2.4 Psychological Justification of Plan Deconstruction

This post-hoc justification of plan formation may seem in some ways unsatisfactory, since we can never be sure that the justification corresponds to the way in which the plan was actually formed. We have shown above that this is going to be a rare problem

that will be solvable even in its worst form. However, this kind of post-hoc reasoning is also interesting because of the way in which it mirrors human reasoning.

[Nisbett and Wilson, 1984] discusses how people generally do not know why they performed certain actions. If challenged, they will subconsciously make up a justification that sounds plausible, and be convinced that this is, in fact, why they performed the action. However, there is no true introspection when people report on their cognitive processes. Instead, their reports are based on a priori, implicit causal theories. Accurate reports occur if it is logical to conclude that the stimuli caused the response, and not otherwise. In a human, there are often complicated psychological reasons behind why people arrive at their explanations; for example subconscious moral justifications for action they feel uneasy about. However, according to Nisbett and Wilson, the constructive processes themselves never appear in consciousness, only their products do. Likewise, in the plan deconstructor, the planner provides only the plan and not the justification for the plan. The justification is reconstructed by the plan deconstructor. As discussed above, this post-hoc justification proceeds along more logical lines than is common in human post-justification of actions.

6.4 Translation

Since we are working with three different representations, it is necessary to be able to translate freely between them. In this section, we describe the process of translation and the logical implications of these translations.

Both of the translation processes are one-way. It is not necessary to translate the whole ontology from PDDL or Prolog back into KIF, because the KIF ontology is maintained as the central ontology. Any ontological changes that occur during this process are also made individually to the KIF ontology at the end of the process. During plan execution, the Prolog ontology is more up to date than the KIF ontology, but as soon

as plan execution terminates, all of this updated information is then passed to the KIF ontology. Refinements are made to the KIF ontology only. After the refinement has been performed, the translation process is performed again so that the changes can be propagated to the PDDL and Prolog representations.

Since the agent communication system is written in Prolog, small translations outside of the two translations of the whole ontology are necessary. There are three types of these:

1. When the goal is passed to the agent, it must be passed in Prolog. It is then translated directly into syntax readable by PDDL (which is coincidentally the same representation as used by KIF), so that it can be passed to the planner. For example,
   attendConference(researcherMcneill,aiConference05)
   would be translated to:
   (Attend-Conference Researcher-Mcneill Ai-Conference05)

2. When the plan is produced, it is written to an output file. In addition to the plan, a lot of extraneous information is produced; for example, number of plan steps, time to produce the plan, and so on. The file is sifted for the plan steps, which are then translated into a format that is readable by the agent (which is written in Prolog).

3. When information from the agent is used to alter the KIF ontology, this needs to be translated into KIF. This happens both during normal updating of the ontology after plan execution terminates, and when refinements are performed. In the former situation, we are dealing only with facts. In the latter, we may be dealing with facts, or we may be altering definitions of predicates, rules, or class hierarchy. Such translations from Prolog to KIF are very small: for example, they may concern the name of a predicate and a new name that it should take, and both

these names would need to have their initial letter changed to uppercase. Further detail of this process is given in Section 5.5.1.

6.4.1 The Top Level Translation Process

Before translation to either PDDL or the Prolog representation begins, some initial processing of the KIF ontology is carried out. First of all, the KIF ontology is read and each line is sorted appropriately. Lines that do not contain ontological objects, for example, blank lines and marker lines, are discarded. Lines that do contain ontological objects are sorted into relevant lists to produce a list of axioms, a list of individuals, and so on. The situational arguments, that describe in which situation a fact is true, are removed in all cases. This situational information is not required by the planner, since it can keep track of the situation implicitly. Nor is it required in the Prolog version: since the actions are not presented as implication rules in Prolog, the situational arguments are not necessary to maintain consistency. Once these lists have been produced, they are then processed further to extract the relevant information, and to remove all the KIF markup in the axiom, which would be irrelevant in PDDL and Prolog. Each axiom in the axiom list is converted into a list containing the axiom name, the preconditions of the action, and the effects of the action. The relation list is sorted so that each relation becomes a list that contains information about its name, what class its arguments are and what order they are in.

The most difficult list to build is the fact list. Facts are not ontological objects in their own right in a KIF ontology. Instead, they are attached to the individual (or one of the individuals) to which they pertain. Thus the fact list is built up simultaneously with the individual list.

At the end of this process, seven lists are produced:

- rule (or action) list

- predicate (or relation) list
- numerical predicate (or function) list
- individuals list
- fact list
- class hierarchy list
- class list (for the problem file)
- class list (for the domain file)

The last two lists contain the same information. They are processed in different ways because different representation is required for the problem file to the domain file. Processing them differently at this level simplifies the process later on. Note that the names *problem* and *domain* refer to the needs of the PDDL translation process; this is because it is here that we need to make the distinction, and these names are used for clarity. Only the problem file class list is passed to the translation to Prolog process. Also, the list of individuals is not passed to Prolog. This is needed in PDDL because every individual must be declared. In the Prolog representation this is not necessary; individuals need only be referred to in the context of facts that concern them, and in their class information, and this is contained in the fact list.

These lists (with the exceptions mentioned above) are then passed to the *translateToPDDL* process and the *translateToProlog* process.

6.4.2 The Meta-Ontology

Once the main ontology has been processed by the top level translation process, the meta-ontology is dealt with. First of all, a list is produced containing all the relevant lines from the ontology. Much of the relevant information concerns individuals,

because in the meta-ontology, objects that in the ordinary ontology are functions, or relations, or actions, become individuals. There are a few relations defined in the meta-ontology: *agentNeeded*, which attaches an individual agent to an individual action, and various unary relations that predicates can take that determine how they are to be interpreted during plan execution. These are discussed in more detail in Section 6.4.4.2. The class hierarchy of predicates is included in the meta-ontology since when we consider the predicates to be individuals rather than relations, we can assign a class to them.

The information contained in the meta-ontology is processed to produce list of predicate names that are attached to descriptions of their behaviour, plus declarations of instantiations of the meta-predicate *agentNeeded*.

6.4.3 Translation From KIF To PDDL

The work discussed in this section has been published in [McNeill et al., 2004a]. As discussed above, the KIF ontology is separated into two parts: the meta-level ontology and the object level ontology. The meta-level ontology contains information for the agent about how to communicate with other agents and how to manipulate the predicates during this process. Because the PDDL version of the ontology is used only for generating plans, and the information received from this process informs the agent only of the plan steps it must execute, there is no need to encode the meta-level ontology in the PDDL ontology: the object-level ontology contains all the information required for the generation of plans.

6.4.3.1 KIF and PDDL Representation

There are six different types of ontological objects in a KIF ontology: functions, relations, axioms, classes, individuals and frames. Note that the term *function* has a

slightly different meaning in KIF to PDDL. In KIF, a function refers to a kind of *relation* (or *predicate*), that, given instantiations for $n-1$ arguments, has a precisely determined value for the nth argument. On the other hand, a PDDL predicate for which the above holds is only referred to as a function if the nth argument is numerical [Fox and Long, 2003]. A KIF *relation* corresponds to a PDDL *predicate*, with the exceptions stated above: PDDL *predicates* include those that are uniquely determined but non-numerical, whereas in KIF, these would be considered to be *functions* and not *relations*. To avoid confusion, we prefer instead to refer to *predicates* and *numerical functions*. A KIF *axiom* corresponds to a PDDL *action*; that is, a rule describing the preconditions and effects of a named action, with the exception that KIF axioms have preconditions and effects that contain situational arguments, and PDDL actions do not. KIF *classes* correspond to PDDL *types*. In the ontologies we have been working with, KIF *frames* and *individuals* both correspond to PDDL *objects*, although in more complex ontologies, *frames* can refer to any ontological object, e.g., classes. Since we have used simple versions of the other ontological objects, it has not been necessary to consider these as *frames*. A *frame* is an individual that has initial facts attached to it; an *individual* has none. The initial status of the problem is extracted from information contained within the frames and individuals of the KIF ontology. The different types of ontological objects allowed in each representation are summarised in Table 6.1.

Our KIF ontologies have been developed using the Ontolingua Ontology editor [Gruber, 1992, Farquhar et al., 1996]. This produces an HTML page containing the whole ontology, which can be saved to a single file. PDDL requires this file to be translated into two files, the *domain* file and the *problem* file (see Figure 6.4 and the example below). In PDDL, the domain file contains information that is general to the whole domain: the names of predicates, the numbers of arguments they take, the axioms, and so on. The problem file contains the information that is specific to a particular prob-

[1] note that in our restricted KIF, frames always translate into PDDL objects. However, this is not generally true with full KIF.

KIF	**PDDL**
Relation	Predicate
Function	Predicate or Function
Axiom	Action
Class	Type
Frames	Objects [1]
Individuals	Objects

Table 6.1: Comparison of KIF and PDDL Objects

lem: the individuals, their classes, the facts and the goal. Hence a single domain file can be paired with many different problem files. In Ontolingua-KIF the whole ontology is contained in a single file. Some types of KIF ontological objects are put in the problem file, and some in the domain file, because the KIF ontology contains not only a description of the domain, but also facts and individuals.

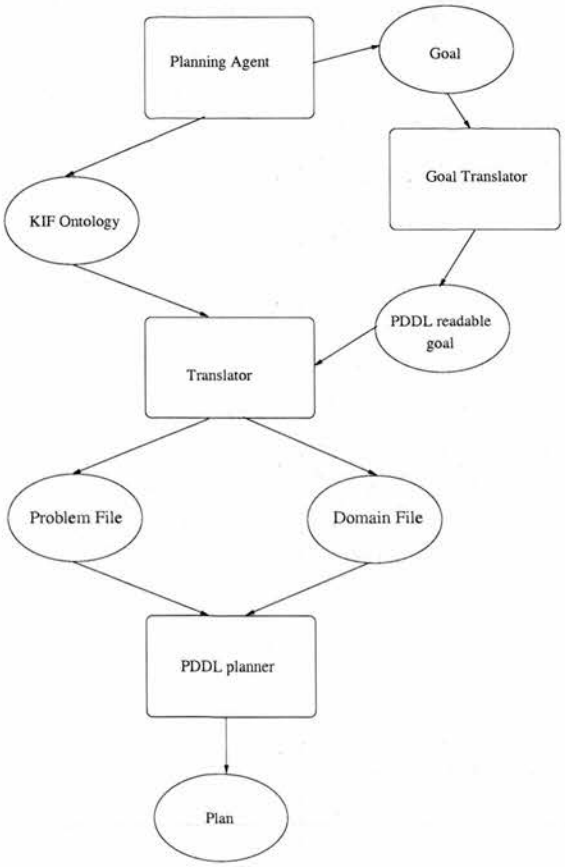

Figure 6.4: Architecture of PDDL Translation System

6.4.3.2 Motivating Example

Consider the situation in which a virtual travel agent is given a goal to purchase an plane ticket online. In order to achieve this goal, several steps must be carried out. For example, the agent must locate a ticket selling agent, it must ensure it has sufficient funds, it must work out the correct origin and destination for the flight, and so on.

Clearly, before the agent can act, it must have a plan for how to achieve the goal. Therefore, as soon as the agent identifies a goal, it sends the whole ontology, together with a suitable representation of this goal, to the translator. PDDL files for the ontology are produced, which can then be sent to the planner. The planner will produce a plan for how to achieve this goal, which can be translated into a format that is readable by the KIF agent. Once the KIF agent has the plan, it can then begin to execute the plan steps.

In this short example, we have the following ontological objects in the KIF ontology:

```
(Define-Frame Travel-Agent :Own-Slots ((Instance-Of Agent))
                           :Axioms ((Money Travel-Agent 500 [(Start)])))

(Define-Frame Edinburgh :Own-Slots ((Instance-Of City))
                        :Axioms ((Flight Edinburgh London 300 [(Start)])))

(Define-Individual London (City))

(Define-Function Flight (?Place-0 ?Place-1 ?Situation) :-> ?Value
                         :Def (And (Place ?Place-0) (Place ?Place-1)
                                   (Situation ?Situation) (Number ?Value)))

(Define-Function Money (?Agent-0 ?Situation) :-> ?Value
                        :Def (And (Agent ?Agent-0)
                                  (Situation ?Situation) (Number ?Value)))

(Define-Class Agent (?X) :Def (And (Thing ?X)))

(Define-Class City (?X) :Def (And (Place ?X)))

(Define-Class Place (?X) :Def (And (Thing ?X)))

(Define-Axiom Book-Flight :=
```

```
(=> (And (Flight ?Origin ?Destination ?Price ?Sit1)
         (Money ?Agent ?Amount ?Sit1)
         (< ?Price ?Amount))
    (And (Has-Ticket ?Agent ?Sit2)
         (= ?Newamount (- ?Amount ?Price))
         (Money ?Agent ?Newamount ?Sit2)
         (Not (Money ?Agent ?Amount ?Sit2)))))
```

Note: there are objects referred to in the axiom that are not defined in the ontology section above. These have been omitted for brevity.

This would produce the following PDDL domain file:

```
(define (domain domain Ont)
  (:requirements :strips :fluents :typing)

  (:types agent city place)

  (:functions
   (Money ?Agent)
   (Flight ?Place1 ?Place2)
  )

  (:action Book-Flight
    :parameters (?Agent ?Origin ?Destination)
    :preconditions (And (< (Flight ?Origin ?Destination) (Money ?Agent))
                        (Agent ?Agent) (City ?Origin) (City ?Destination))
    :effects (And (Has-Ticket ?Agent)
                  (decrease (Money ?Agent) (Flight ?Origin ?Destination)))
  )
)
```

and the following PDDL problem file:

```
(define (problem problemOnt)
 (:domain domainOnt)
 (:objects London Edinburgh Travelling-Agent)
 (:init
  (Agent Travelling-Agent)
  (City London)
  (City Edinburgh)
  (= (Money Travelling-Agent) 500)
  (= (Flight Edinburgh London) 300)
 )
 (:goal
   (Has-Ticket Travelling-Agent)
 )
)
```

6.4.3.3 Translation Process

The agent wishing to form a plan will already have (or will prompt for) a goal. As shown in Figure 6.4, this goal will then be passed, together with a file containing the KIF ontology, to the translator. The translator produces two files, the domain file and the problem file, as discussed above. Figure 6.5 illustrates the steps and dependencies involved in translation, which are discussed in more detail below.

The translator is written in Prolog and works largely through pattern matching. For example, a key predicate is the *matchExpression* predicate, which takes a range of characters and an identifier that may or may not appear within that range and, if it finds the identifier, returns what comes before and after that identifier, and otherwise fails:

matchExpression(-BeforeIdentifier,+Identifier,-AfterIdentifier,+Range)

In the above expression, following the Prolog convention, + indicates that this argument is instantiated when the predicate is called and - indicates that this predicate is

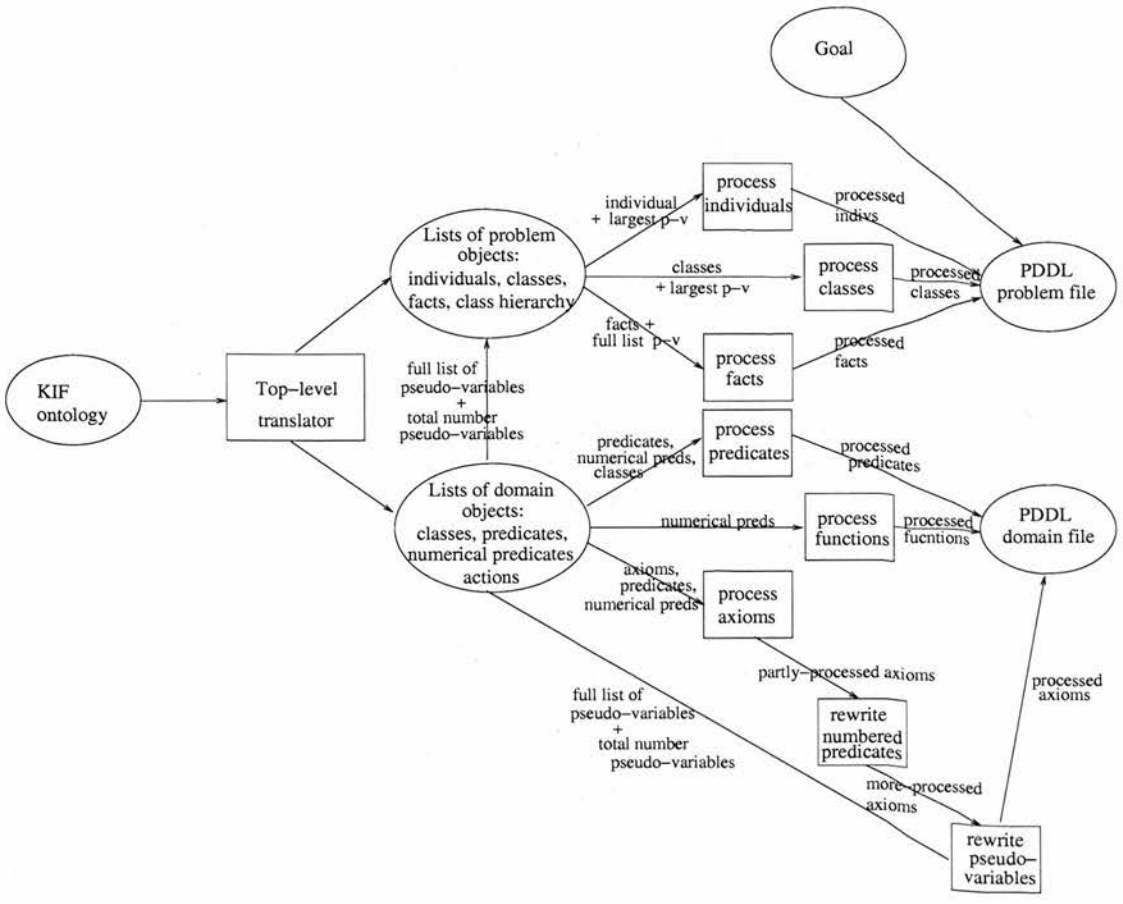

Figure 6.5: Translation Process

uninstantiated when the predicate is called and is instantiated by the predicate. That is, *matchExpression* is passed an identifier and a range of code, and returns what comes before that identifier and what comes after. For example:

matchExpression(Before,'Instance-of',After,'(Instance-of Agent)').

will return:

Before = '('

After = ' Agent)'.

Dealing with Numerical Functions

The most significant difference between KIF and PDDL is the way that numerical functions are dealt with. The example ontology in 6.4.3.2 illustrates that the way in

which KIF functions are defined does differentiate them from KIF predicates. The arguments of a function are defined, e.g., *(?Place-0 ?Place-1)* :→ *?Value* rather than simply as *(?Place-0 ?Place-1 ?Value)*. That is, in a function definition, n arguments are mapped to a single argument, whereas a corresponding predicate definition would simply have $n+1$ arguments. (Note that it is coincidental that *?Value* is numerical: a KIF function would be defined in such a way even if it did not contain numerical arguments). However, when a numerical function is referred to in a KIF ontology, either within an action or as an initial fact, it is dealt with not as a function but as a predicate. For example, a numerical function might be described as:

(Define-Function Money (?Agent) :→ *?Amount)*,
that is, as a function, but a possible instantiation would be:
(Money PA 100),
so that it looks like a predicate.

In PDDL, the numerical argument is not included in the predicate definition, but rather it is written as a function, so that it would be stated:
(Money ?Agent)
and would appear within the *function* definitions in the domain file rather than in the *predicate* definitions.

The specific value of this function is not explicitly mentioned. The PDDL planner would be aware that this had a numerical value attached to it because it would be declared within *functions* rather than within *predicates*. Although it appears that information has been lost here, this is not, in fact, the case. The value of the function is tracked implicitly by PDDL; thus the information remains but it is no longer explicitly represented. If there is an instantiation for this numerical function in this initial state, then the value of this would be stated as follows:
(= (Money Agent) 100).

During the planning process, the PDDL planner will keep track of the value of all the

functions and these changing values are not referred to specifically within the axioms. However, in a KIF axiom, these values must be referred to and are thus given explicit names. For example, a *Buy* rule may have a precondition that the amount of money the buying-agent has must be greater than the cost of the item which is purchased. In KIF this would be stated as follows:

(Money ?Agent ?Amount) $\wedge$ *(Cost ?Item ?Price)* $\wedge$ *(> ?Amount ?Price)*

whereas in PDDL, this would be stated as:

(> (Money ?Agent) (Cost ?Item))

An effect for the same action might be that the money that the agent now has is the original amount less the cost of the item. In KIF, this would be:

(= ?Newamount (- ?Amount ?Price)) $\wedge$ *(Money ?Agent ?Newamount)* $\wedge$ *(Not (Money (?Agent ?Amount)))*.

In PDDL, this would be:

(decrease (Money ?Agent) (Cost ?Item))

Dealing with this difference in representation for numerical predicates is by far the most difficult aspect of the translation process. It causes some difficulties in writing the problem file, though these are not particularly hard to solve. More complex are the difficulties this creates in writing the domain file, and particularly in the statement of the axioms. These problems are discussed below.

Writing the Problem File

A PDDL problem file contains the specific details of this particular problem within the domain described in the domain file. The input for this process is the goal, and the list containing all the KIF definitions relevant for the problem file, which are those pertaining to individuals. The PDDL problem file needs to contain:

- A list of the names of the individuals

- A list of what is true initially, which includes:
  - A list of the classes of individuals
  - A list of the initial facts; *i.e.*, initial instantiations of the predicates
- The goal

In KIF, facts are not stated independently but instead are attached to the first individual to which they pertain. For example, *(Location Agent1 Timbuktu)* would be contained within the definition for the individual *Agent1*.

The individual information contained in the list sent to the problem file has not been processed at this stage, merely sifted for information relevant to the problem file. All the definitions within this relevant list are exactly as they appear in the KIF ontology. The first step is to process this list by extracting the useful information from the KIF definitions and forming it into three lists that correspond to the three items listed above (excluding the goal). This is done by searching for key markers within the definition. For example, an individual will either begin with the statement *Define-Individual*, if there are no facts attached to this individual definition, or *Define-Frame* if there are attached facts. The name of the individual always appears immediately after this initial marker. If the marker is *Define-Individual*, we need only extract the class of this individual. If the marker is *Define-Frame*, we then need to find the facts attached to this individual.

Examples are given below:

```
(Define-Individual Isabelle-Paper-Dvi (Dvi-Paper))
```

This line, when processed, adds *Isabelle-Paper-Dvi* to the list of individuals and *(Dvi-Paper Isabelle-Paper-Dvi)* to the list of classes.

```
(Define-Frame Lucas :Own-Slots
  ((Has-Paper Isabelle-Paper-Dvi)
   (Instance-Of Agent)
   (Location Edinburgh))
  :Axioms ((Money Lucas 1000)))
```

This line adds *Lucas* to the list of individuals, *(Agent Lucas)* to the list of classes and *(Has-Paper Lucas Isabelle-Paper-Dvi)* and *(Location Lucas Edinburgh)* to the list of facts. The fact *(Money Lucas 1000)* is also extracted from this line. However, because *Money* is a numerical function, it requires further processing, and *(= (Money Lucas) 1000)* is added to the list of facts.

Once the entire list of relevant definitions is processed, the lists containing this information, together with the goal, are passed to a predicate which writes the problem file. This first writes the necessary initial information, such as the name of the problem that is being defined and the name of the domain within which the problem is described, to the problem file. The three lists (of individuals, classes of individuals and facts) are processed by simply writing them member by member within the correct brackets and initialisers. Finally, the goal, which has been translated from the Prolog format in which is was input to a format readable by PDDL, is inserted into the correct place.

Writing the Domain File

The domain file contains:

- Predicates, which includes:
  - all predicates that do not have a numerical value
  - class names
- Functions (numerical predicates)
- Actions, which contain the following information:

- a list of all the variables mentioned in that action
- the preconditions of the action
- the effects of the action

The relevant lines of definitions are those defining KIF functions, relations, axioms and classes. As discussed above, KIF functions do not correspond directly to PDDL functions, because PDDL only considers KIF numerical functions to be functions; non-numerical functions are considered to be predicates. KIF axioms correspond to PDDL actions. These input lines are processed to create four lists of information required by the domain file: a list of all the predicates (this includes both numerical and non-numerical predicates, *i.e.*, both KIF relations and KIF functions, and both PDDL predicates and PDDL functions), a list of the classes, a list of the actions and a list of the numerical functions. The format of the list of all the predicates and the list of the numerical functions is different, because the former are represented as predicates and the latter as functions. In the latter, the predicates are listed with the numerical argument removed (which is how they must be expressed in PDDL; see above), whereas in the predicates list, because they are not identified as being numerical, each is listed as a predicate name, followed by a list of all the arguments and their classes. In the list of actions, each action is stored as an action name followed by a list containing all the preconditions, as they appear in the KIF ontology, and all the effects.

Writing the domain file is more complex than writing the problem file, largely due to the difficulties with actions, which are discussed below. The file is initialised by stating the name of the domain file and the PDDL requirements. The predicate and class lists are adapted without too much difficulty so that they can be written down in the appropriate place. The numerical function list is used to write down the functions. Note that in our system at the moment, numerical functions are written down both as predicates and as functions. In the former version they have an extra argument (the numerical argument) which is not included when they are written as functions. It is

fairly trivial to check predicates against the numerical predicate list and only write down those that are not numerical in the ordinary predicate slot. However, this is not done for reasons discussed in the discussion of pseudo-variables below. Expressing these numerical functions twice in different ways and in two different definition areas does not raise problems, as the planner considers them to be two different objects. It is never possible that both the function and the predicate version can be used in planning, and so this does not lead to confusion in planning. A more detailed explanation of this is given after the use of pseudo-variables has been explained.

Pseudo-Variables

One of the limitations of PDDL is that it cannot deal with uninstantiated variables. This is because, although PDDL appears to be a first-order language, most PDDL planners are in fact only pseudo-first-order, and work by creating all possible instantiations of the problem and searching through them in a propositional manner. This is a problem for our system, as we wish to deal with agent plans in which there are unknowns after planning. For example, an agent may have a plan to attend a conference which involves registering at the conference and thereby receiving a registration number, and then using that number when actually attending the conference. Such confirmation numbers are useful in an agent system, as they allow the tracking of external objects that the agents possess, or privileges to which they are entitled. When forming a plan, it is not necessary, and indeed it is impossible, to know what these confirmation numbers are. These can only be instantiated during plan execution.

In order to force PDDL to deal with these uninstantiated variables, we have developed a class called *Confirmation-Number* and an individual belonging to that class called *Pseudo-Variable*. When writing an ontology, if we are creating an axiom in which a particular variable cannot be instantiated until plan execution, the individual *Pseudo-Variable* is inserted in place of this variable. This variable may or may not be numerical; that is, this pseudo-variable will sometimes be found in predicates that

PDDL considers to be ordinary predicates, and sometimes in predicates that PDDL considers to be functions. However, if we are using *Pseudo-Variable* as a place holder in a predicate in some action, we do not want this predicate to be considered to be a function, since this means that PDDL will expect to be able to assign a specific numerical value to it.

When we are dealing with numerical functions, we either want them to be considered as ordinary predicates, if the numerical argument is replaced by *Pseudo-Variable*, or as functions if it is not. The difficulty is that these *Pseudo-Variable* markers do not appear in the definition of the predicates, but only within specific actions. It is impossible to tell from the definition of a numerical function whether we will want to deal with it as a predicate or as a function. For this reason, since it does not create a problem with the planner, we define numerical functions as both predicates and functions (with one fewer argument), thus allowing PDDL to consider them as either, depending on the axiom it is currently dealing with. We can, moreover, always be sure that either the function definition is always used and the predicate definition is never used, or vice versa, and thus be sure that no confusion will arise in the planning due to this double definition. Any function or predicate can only be used in planning if there is an instantiation of it. Thus there must either be an instantiation of it in the initial situation, or it must be made true as an effect of an action. A numerical predicate that requires a pseudo-variable can never exist in an instantiation with a numerical argument, and vice versa, because neither will it be instantiated in this way in the initial state, nor will it be instantiated in this way as the effect of an action. Thus, for a numerical predicate that requires a pseudo-variable, the function declaration of it will never have an instantiation, and thus cannot be used in planning, and vice versa.

It is also necessary to number pseudo-variables, otherwise the planner will believe that they all refer to the same individual. Every time a *Pseudo-Variable* is used to refer to a specific thing, the same identifying number must be used. Thus if an effect of

an action is *(Has-Ticket Agent Pseudo-VariableX)*, then if this same predicate appears as a precondition to a different action, the identifying number of the *Pseudo-Variable* must again be *X*; otherwise PDDL will consider the two pseudo-variables to refer to different individuals, and the effect of the first rule cannot be used as a precondition to the second rule. To facilitate this, the preconditions and effects of each action are processed as normal, with each numerical variable that is to be treated as non-numerical being named *Pseudo-Variable*. When all the axioms have been processed in this manner, the pseudo-variables are then updated. Firstly, the preconditions of each action are processed, adding an identifying number to each occurrence of *Pseudo-Variable*. The name of each predicate in which this pseudo-variable occurred is recorded, together with the identifying number for that predicate and the other arguments of that predicate, so that each identified *Pseudo-Variable* can be reliably tied to the correct predicate. The effects are then processed, attaching the correct identifying number to each *Pseudo-Variable*. The above processing also returns the number of *Pseudo-Variables*. Thus when the problem file is written, this number of *Pseudo-Variable* individuals can be declared and their classes (*Confirmation-Number*) given.

During the cyclical process of the system, it may be that a new fact is asserted which instantiates a pseudo-variable; this will happen if an action is performed for which this new fact is an effect. When the updated ontology containing this new fact is translated into PDDL, the translator would naturally translate this into a numerical predicate; for example:

(= (Has-Ticket Agent) 125),

where *125* is the instantiated pseudo-variable. However, this way of representing this predicate will clash with the way in which it is represented in the actions, since there it is represented as non-numerical. We do not wish to change the representation of the actions, since we wish these to be applicable to any situation: we may wish to book more flights with unknown reference numbers, and we still need to represent these as

non-numerical. Therefore, this fact is represented as if it were non-numerical:

(Has-Ticket Agent Pseudo-VariableX).

We use the list of pseudo-variable identifiers and the predicates to which they are attached, which is built up when processing the domain file as discussed above, to decide what the value of X should be. If this does not tally with the identifying number used in the rules, then the fact *(Has-Ticket Agent Pseudo-VariableX)* cannot be used to fulfil precondition *(Has-Ticket Agent Pseudo-VariableY)* of an action.

It may seem that by removing the numerical value of this predicate and replacing it with a pseudo-variable, we are removing pertinent information from the ontology. However, it must be remembered that the PDDL representation is used only in planning; the Prolog representation is used during agent interaction, and the KIF representation is the definitive representation. It is precisely because the value of the pseudo-variables is not relevant in planning, but only in plan execution, that we are able to use them to replace variables in this manner. Thus it is not important that this information is lost to the planning representation.

Meta-variables

After refinements have been performed, it may be the case that the KIF ontology contains facts with meta-variables in them. This situation cannot be replicated in the PDDL version of the ontology, because variables are not permitted in PDDL. In order for the planner to be able to produce a plan, it is essential that the meta-variables are instantiated.

We solve this problem by finding all the objects of the correct class, and picking one of them to replace the meta-variable in the PDDL representation. The object is chosen by observing which object was referred to by the agent which caused the refinement: propositional anti-abstraction is diagnosed when the PA encounters an agent using a version of a predicate in its ontology which has a higher arity than expected. The

way in which the agent instantiates this argument is taken to be the way in which this argument should be instantiated in our ontology. It should be emphasised that this choice of specific argument only occurs in the PDDL representation. In the KIF ontology, this remains as a meta-variable until the instantiation has been successfully used during plan execution. If the instantiation is not used in planning, or if it is used in a plan that fails during execution, it remains as a meta-variable in the KIF ontology. For example, if a predicate *(Money ?Agent ?Amount)* is refined to *(Money ?Agent ?Amount ?Currency)* because a question was asked about *(Money AgentX ?Amount Dollars)*, then all money facts in the KIF would be altered to *(Money Agent Amount Meta-Var)* (with *Agent* and *Amount* instantiated appropriately). However, when such facts were translated, the variable could not remain if they were to be used by the planner. Thus the known instantiation is used, and the facts would become, in the PDDL version, *(Money Agent Amount Dollars)*.

This solution is clearly unsatisfactory. We cannot know what the correct way to instantiate any of these arguments is, and, in choosing this course, we may be introducing further error into our ontology. On the other hand, by using this approach, we at least have the option to keep on planning with our ontology, which would be impossible otherwise. It may be better to allow the translation process to choose how to instantiate each argument, rather than assuming that it will be instantiated in the same way as the other agent instantiated it. It is clear that during that particular interaction we would need it to be instantiated in that way, but other occurrences of it may be instantiated differently. This would be possible by adding multiple occurrences of the fact to the PDDL representation, each one instantiating the extra argument differently so that every possible combination was tried. However, this would imply that all of these possibilities were true simultaneously, which would not be the case. It is not possible in PDDL to represent the idea that exactly one of many options is possible; once one has been chosen, the rest are no longer possible. The best way to simulate this would be to keep in the KIF ontology information about which instantiations had been tried,

and if plan failure occurs in a way that implies this fact may have been at fault, another instantiation is tried. However, this is not currently implemented in ORS.

Creating Actions

One of the more difficult tasks involved in writing the domain file is dealing with the numerical functions within the actions. In action definitions, it is not simply a case of inserting definitions. Instead, we must sometimes deal with arithmetic operations. An example of a KIF rule containing arithmetic operations, and its PDDL equivalent, are given below:

KIF rule:

```
(Define-Axiom Buy ''this describes a buy action'' :=
  (=> (And (Price ?Item ?Cost)
           (Money ?Agent ?Amount)
           (Location ?Agent ?Shop)
           (< ?Cost ?Amount))
      (And (Has ?Agent ?Item)
           (= ?Newamount (- ?Amount ?Cost))
           (Money ?Agent ?Newamount)
           (Not (Money ?Agent ?Amount)))))
```

PDDL rule:

```
(:action Buy
  :parameters (?Item ?Agent ?Shop)
  :preconditions: (And (< (Price ?Item)
                          (Money ?Agent)
                          (Thing ?Item)
                          (Agent ?Agent))
                       (Location ?Agent ?Shop)
  :effects: (And (decrease (Money ?Agent)
```

```
                    (Price ?Item))
                (Has ?Agent ?Item))
)
```

The first step is to alter the logic of the KIF to bring it in line with the logic of PDDL. That is, turn the KIF predicates into functions by removing the explicit representation of the value. For example, the precondition

(Money ?Agent ?Amount)

would be rewritten to the precondition:

(Money ?Agent).

It appears that information has been lost in this process. However, the information contained in the variable *?Amount* still exists, it is just not explicit. PDDL tracks the values of all of the functions: a value will have been declared for *(Money ?Agent)* either initially or in a previous rule. The value contained in *?Amount* will be assigned implicitly to the PDDL function, and thus there is no need to represent it explicitly. However, we cannot immediately forget about the variable *?Amount*, because this will be used at other stages of the rule to refer to the value of *(Money ?Agent).* It is still necessary to link these functions to the variable that represented their value, so that we know how these should be replaced within the arithmetic. *?Amount* is a marker for the value of *(Money ?Agent)*, and one can always refer to *?Amount* at any place in the KIF preconditions or effects of that action, and this will be a reference to the value of *(Money ?Agent).* Thus if we wish to change the amount of money, we can change the value of *?Amount* and assert this as the new argument of the predicate:

(= ?NewAmount (- ?Amount ?Cost)) ∧ (Money ?Agent ?NewAmount) ∧ (Not (Money ?Agent ?Amount)). When we treat these predicates as functions, we lose this value marker. In PDDL, it is not necessary to have a marker for the value of a function, because these values are automatically tracked by the planner. However, when we are translating to PDDL, we need to keep a record of these markers so as to be able to determine where these new functions should be placed.

We perform the transformation:

$(f\ ?\vec{x}\ ?y) \wedge \Phi \Rightarrow \Phi\{?y/(f\ ?\vec{x})\}$

In the above expression, f indicates a function, $?\vec{x}$ indicates one or more variables, $?y$ indicates a single variable and Φ indicates the whole of the preconditions and effects. $\Phi\{?y/(f\ ?\vec{x})\}$ indicates the preconditions and effects, with every occurrence of $(f\ ?\vec{x})$ replaced by the variable $?y$; $?y$ is the marker for the function $(f\ ?\vec{x})$.

The first thing to be done is to strip all the predicates that will become numerical functions from the rule, keeping a record of their markers, and then replace any occurrence of these markers with the numerical functions. For example:

:preconditions (And (Money ?Agent ?Amount) (Price ?Item ?Cost) (< ?Cost Amount)
(Location ?Agent ?Shop))

:effects (And (= ?NewAmount (- ?Amount ?Cost)) (Money ?Agent ?NewAmount)
(Not (Money ?Agent ?Amount)))

would first of all become:

Preconditions: (And (< ?Cost ?Amount) (Location ?Agent ?Shop))

Effects: (And (= ?NewAmount (- ?Amount ?Cost))),

with stored information:
[?Amount(Money ?Agent), ?Cost(Price ?Item), ?NewAmount(Money ?Agent)]

The role in KIF of these predicates that have been removed is to create an identifier for the value. That is, by stating *(Money ?Agent ?Amount)* in the KIF preconditions, we have declared that *?Amount* is the temporary name given to the amount of money that *?Agent* has. Note that we now have two different markers for the numerical function *(Money ?Agent)*, because the value of this function is changed by the rule. In KIF, there is no problem with having the same predicate with different markers, as

the markers distinguish them. However, if we were to replace both these markers by the functions to which they are attached, we would have two occurrences of the same function, *(Money ?Agent)*, which would each time take a different value. For example, this would lead to statements such as:

(= (Money ?Agent) (- (Money ?Agent) (Price ?Item)))

which, since *(Price ?Item)* has a non-zero value, is not logically consistent. The reason these inconsistencies occur is because we have, at this stage, changed the logic but not changed the syntax. Since these predicates have now become functions, we have no need to assign values to them in the previous manner: we do not need an equality statement. For this reason, we do not replace markers that come immediately after an equals sign. Instead, we leave them in for this stage of the rewriting, and remove them later when the syntax is altered. So, after we have replaced the markers with the numerical functions, we have:

Preconditions: (And (< (Price ?Item) (Money ?Agent)) (Location ?Agent ?Shop))

Effects: (And (= ?NewAmount (- (Money ?Agent) (Price ?Item))))

with stored information:

[?NewAmount(Money ?Agent)]

We now need to alter the syntax so that is also in line with PDDL. There are three different types of operators that we need to consider: comparative operators, arithmetical operators and assignment operators. For comparative operators, the syntax of KIF matches the syntax of PDDL: once we have replaced the markers with the functions, we already have a readable PDDL comparator:

(< (Price ?Item) (Money ?Agent))

However, arithmetical and assignment operators are rather more complex. In KIF, assignment operators are always denoted by an equals sign, and the manner in which the assignment is being made is contained within the equality. For example,:

(= ?NewAmount (- ?Amount ?Cost))

means assign to the variable *?NewAmount* the value of *?Amount* less the value of *?Cost*. The arithmetical operator - gives further information about the way in which the value is assigned: in order to find the value of *?NewAmount*, we decrease *?Amount* by a certain amount. In PDDL, there are five assignment operators: *assign, scale-up, scale-down, increase* and *decrease*. So an expression in KIF that requires two arithmetical operators, = and -, can be represented in PDDL by a single operator, *decrease*. Likewise, an equality statement containing a + would correspond to *increase*, one containing a * would correspond to *scale-up*, and one containing a / would correspond to *scale-down*. We use these four assignment operators, as opposed to simply *assign*, because the function to which the value is being assigned is the same as one of the functions in the arithmetic expression: in this case, we are finding a new value for *(Money ?Agent)* by altering the old value by the amount represented by *(Cost ?Item)*. However, if we are assigning a value to a different function, we use *assign*.

In our above example,

(= ?NewAmount (- ?Amount ?Cost))

will eventually become:

(decrease (Money ?Agent) (Price ?Item)).

However, if the variable that was being assigned a value (in this case *?NewAmount*) did not correspond to a function within the equality statement, we would use *assign*. For example, if *?NewAmount* was a marker for a function *(Random-Value)*, the above statement would be converted to:

(assign (Random-Value) (- (Money ?Agent) (Price ?Item)))

or perhaps *?NewAmount* refers to the money of another agent. We would then have:

(assign (Money ?Agent1) (- (Money ?Agent) (Price ?Item))).

In this situation, because the arithmetical operator is not contained within the assignment operator, as it is in *decrease*, it must be used explicitly.

Sometimes, KIF statements assign values to variables that do not correspond to functions at all. For example:

Preconditions: (And (Price ?Item1 ?Cost1) (Price ?Item2 ?Cost2)
(Price ?Item3 ?Cost3) (Money ?Agent ?Amount))

Effects: (And (= ?Total (+ ?Cost1 ?Cost2 ?Cost3))
(= ?NewAmount (- ?Amount ?Total)) (Money ?Agent ?NewAmount))

This is similar to the preconditions and effects of the rule above, except that we have a variable *?Total* which is a place holder for an expression, rather than a marker for a function. This is dealt with in a similar way to the function markers. The variable *?Total* is removed from the expression but information about what it is referring to is retained. It can then be inserted into the statement at a later stage. This would eventually lead us to:

Preconditions: (And ())

Effects: (And (decrease (Money ?Agent) (+ (Price ?Item1) (Price ?Item2)
(Price ?Item3))))

However, this would still not be correct PDDL. In KIF, the arithmetic function + can take two or more arguments, whereas in PDDL, + can only take exactly two arguments. Thus, if we find a + expression with more than two arguments, they must be nested. So the effects would become:

Effects: (And (decrease (Money ?Agent) (+ (Price ?Item1) (+ (Price ?Item2)
(Price ?Item3)))))

We have similar problems with the other arithmetical operators, and they are dealt with in a similar manner.

Although there are certain complications with the translation of preconditions and effects, some of which have been discussed above, it is nevertheless relatively straightforward to show that every case has been considered. There are a small number of KIF operators which correspond to a small number of PDDL operators and so, once the translation of some has been implemented, it is not difficult to generalise it so that it can apply to any KIF arithmetical statement.

Once the preconditions and the effects have been processed, all that remains to be done is to identify the variables used in the action, so that these can be declared. This is done simply by building a list of variables by stripping all the variables from the processed preconditions and postconditions, and then removing any duplicates from this list. This must be done after the preconditions and effects have been processed, as otherwise we will declare variables that do not appear in the processed preconditions and effects, such as *?NewAmount* or *?Total*.

Once these three lines of information — the variables (parameters), the preconditions and the effects — have been developed, the action can very easily be written down in the correct place in the file. All that remains is to locate the name of the action and place that in the proper place.

In summary, the main changes that need to be made are:

- Remove the numerical arguments from KIF numerical predicates, so that the predicate is folded into a PDDL function;
- Remove all occurrences of that numerical predicate that do not appear in an arithmetical expression from the rule; these are there to assign values to the predicate, and are not necessary for PDDL functions;
- Replace all occurrences of the marker (the name of the numerical variable in KIF) with the PDDL function;

- Rearrange the arithmetic and the assignment operators accordingly.

6.4.4 Translation From KIF To Prolog

Ontolingua already has a built-in translator from KIF to Prolog that we could have made use of. However, the output of this translation process is not particular satisfactory [da Silva et al., 2002] and would have required much post-processing to produce an appropriate format. Since the most suitable output for our system has fairly specific requirements, we therefore decided that it would be better to produce our own translator.

The translation to Prolog is significantly less complicated than the translation to PDDL. In the latter process, we had to conform to a representation that was not specified by us; any deviation from this would result in a representation that could not be considered PDDL. However, when translating to the Prolog representation, the only constraints is that the result must be readable by an agent written in Prolog. Beyond that, we are free to choose a representation that fits in easily with the KIF translation, since this representation is only for the internal workings of the agent communication system, and is not required to conform to any other specifications.

In the plan deconstructor, the Prolog representation is also split into two files. The *signature file* contains the class hierarchy, the information about predicates and the information from the meta-ontology; the *theory file* contains the initial facts, class information for each individual and the actions, corresponding to the PDDL domain and problem files.

6.4.4.1 Writing the Theory File

The theory file contains the facts, the class definitions and the information concerning which agent is required for each action. The agent information comes from the meta-

ontology and is represented as a Prolog fact:

agentNeeded(agentX,action1).

The facts are extracted from the list of facts that was created during pre-processing, and are expressed as follows:

fact(hasItem(agentX,item)).

The classes are extracted from the class list and are expressed as follows:

class(agentX,agent).

Unlike in PDDL, in the Prolog representation, the superclasses of objects are not expressed explicitly; rather, this information exists in the class hierarchy contained in the signature file. Thus, since *Agent* is a subclass of *Thing*, we would list, in PDDL, not only the above fact, but also the information that *agentX* was of class *Thing*. In the Prolog representation, this is omitted.

The major part of the translation is just to convert the items in the processed lists into a format that can be read by Prolog. In both KIF and PDDL, a variable is indicated by a question mark preceding a name with an uppercase initial letter, e.g., *?Variable*, and a constant is indicated by a name with an uppercase initial letter without a preceding question mark, e.g., *Constant*. In Prolog, however, variables are indicated by names with uppercase initial letters, and constants are indicated by names with lowercase initial letters. Thus *?Variable* must be converted to *Variable*, and *Constant* to *constant*. There are many other syntactical changes that must be made; for example, replacing spaces between arguments with commas, but these are all quite simple to implement. The markup must also be added; for example, surrounding a fact with a *fact(...)* predicate.

Meta-Variables

In the Prolog representation, it is not necessary that the meta-variable place holders be instantiated. In fact, this is undesirable, since we wish to leave the choice as to how to instantiate these until we are forced to make it. This will not cause the Prolog representation to become out of sync with the plan formed by the PDDL representation, which instantiates all of these variables. If one of these instantiated variables is used in the planner, this will be reflected in the output plan, and thus this will cause the uninstantiated variable in the Prolog representation to be instantiated in the same way. Thus we leave these variables uninstantiated, to be instantiated either in the way forced by the plan if this occurs, or through further information gleaned through agent communication. We therefore require only that when the conversion of the names takes place, any individual with the name *Meta-Var* is left with an uppercase initial letter.

6.4.4.2 Writing the Signature File

Writing the signature file is more complex than writing the theory file because it contains the actions, which are the most complex of the ontological objects. These are represented in the Prolog representation in the following way:

```
rule(actionName(Var1,...,VarN),
     [ruleNo,
      [list of preconditions],
      [list of effects]
     ]
    ).
```

The signature file also contains the class hierarchy, as well as the meta-information about predicates and actions, which includes the following information:

- *myFacts list*: a list of predicates about which the PA has authority. This does not mean that they cannot be wrong or that the PA cannot investigate how it came to

believe them; rather, it means that the PA cannot directly ask another agent what their value is, but instead should have some information about that itself. This includes predicates such as *money* and *has*.

- *nonFacts list*: a list of items that are not stated as facts. This includes such things as *class* and *member*. When the PA is processing preconditions and effects, the items mentioned in this list need to be treated differently to ordinary predicates.

- *inform list*: a list of predicates which, if they appear as postconditions, require the PA to ensure that the service-providing agent has given out some information. For example, *registered(PA,thisConference,RegistrationNo)* would require information from the service-providing agent, so that *RegistrationNo* could be instantiated properly.

- *agentNeeded* information. For every action, information concerning which service-providing agent should be contacted to perform it is listed in the form: *agentNeeded(actionX,agentY)*.

The class hierarchy is expressed as a list of subclass relations, for example:

subclass(city,place),
subclass(place,thing).

All classes are ultimately a subclass of thing. If this is stated explicitly in the class hierarchy, it means that it is a direct subclass (that is, a class with thing as its immediate superclass).

The information necessary for forming the rules and the class hierarchy come from the main ontology, and is part of the processed lists created during the top level translation process. The information concerning the predicates and the actions, listed above, comes from the lists created from the meta-ontology.

A great deal of the processing required for the translation to PDDL is reused for the translation to Prolog. However, the most difficult part of that — dealing with numerical functions — is not required for the translation to Prolog, as we choose a representation for the numerical functions that is very similar to how they are expressed in KIF. We use the techniques from translation to PDDL to find all the preconditions and effect of each rule, and, again using translation to PDDL processes, extract the class of each variable so that it can be declared as part of the conditions. If this is not done, the variables will not be bound and may be instantiated incorrectly. Additionally, any condition that contains a calculation is wrapped in a *calculation* predicate, so that the agent can deal with it properly.

Thus preconditions:

(And (Money ?Agent ?Amount) (Price ?Item ?Cost) (< ?Cost ?Amount) (Location ?Agent ?Shop))

would become:

[money(Agent,Amount),price(Item,Cost),calculation(Cost < Amount), location(Agent,Shop),class(Agent,agent),class(Item,item),class(Shop,shop)]

Additionally, the meta-information concerning the *inform* list is used during the processing of the conditions. Any predicate that is a member of the *inform* list will be wrapped in an *inform* predicate if it appears as a postcondition:

(And (Registered ?Agent ?Conference ?Pseudo-Var) ...)

will become:

[inform(registered(agent,conference,Pseudo-Var)),...]

These *inform* predicates appear exactly when there are pseudo-variables present. The purpose of the information received from *inform* predicates is to instantiate the pseudo-vars. If *inform* predicates appear in the preconditions of an action, then they are not

wrapped in an *inform* predicate. They can only appear in the preconditions of an action if they are either fully instantiated, or if the value of the instantiated is immaterial to the action. It is not possible to receive information about one of the preconditions of an action from the service-providing agent; the PA ought already to know how this precondition should be instantiated before the action is performed. Despite the fact that the planner considers pseudo-vars to be instantiated, we can be sure that it will never happen that a predicate which has a pseudo-var that cannot yet be instantiated is used as a precondition to an action. This is because if there is no information about that predicate as a fact in the initial ontology (in which case there must be an instantiation for the pseudo-var), then this predicate cannot be used as a precondition because no instantiation of it holds in the current state. Thus, if it is to be used as a precondition, it must have been made true by being the postcondition of a previous action. But if it appears as a postcondition, it will be wrapped in an *inform* predicate, and the PA will have already discovered how it should be instantiated.

Pseudo-Variables

Pseudo-variables in KIF are less of a problem when translating to Prolog, as opposed to when translating to PDDL, since Prolog can deal with variables and we do not have to pretend that these are instantiated. It would thus be perfectly feasible to call them by any convenient name, such as *RegistrationNo*. However, because the actions are processed centrally before being passed to the translation to the PDDL process and the translation to the Prolog processes, to avoid repeating processing, they are already referred to as Pseudo-Vars when passed to the Prolog translation. We thus leave them referred to as such. In Prolog, it is not a problem if several different variables are referred to by the same name, as long as these variables do not appear in the same clause. Thus, if we have:

```
rule(submitPaper(Conference,PaperPs,Agent),
      [rule7,
      [hasPaper(Agent,PaperPs)],
      [inform(acceptedPaper(Agent,PaperPs,Conference,PseudoVar))]
```

```
    ]).

rule(bookAccom(Conference,MetaVar,Agent),
     [rule3,
      [accommodationInfo(Conference,Cost)],
      [inform(hasAccom(Agent,Conference,PseudoVar))]
    ]).
```

(note that many of the conditions, including the class declarations, are removed here for the sake of clarity)

then Prolog will not expect the two references to PseudoVar to refer to the same object. This contrasts to PDDL, where if PseudoVar appears anywhere in the domain or problem file, then another reference to it anywhere else would be expected to refer to the same objects. However, if two references to a certain name are found within a clause, then these would be expected to refer to the same object. Thus:

```
rule(findAccomInfo(Conference,Paper,Agent),
     [rule2,
      [acceptedPaper(Agent,Paper,Conference,PseudoVar)],
      [accommodationInfo(Conference,PseudoVar)]
    ]).
```

could not be executed correctly, because the two pseudo-variables would be expected to refer to the same object, but in fact do not. We therefore number the pseudo-variables within each predicate. Thus the above examples of *submitPaper* and *bookAccom* would be left unchanged, since each rule definition contains only one reference to PseudoVar. *FindAccomInfo*, on the other hand, would be numbered thus:

```
rule(findAccomInfo(Conference,Paper,Agent),
     [rule2,
      [acceptedPaper(Agent,Paper,Conference,PseudoVar0)],
       [accommodationInfo(Conference,PseudoVar1)]
    ]).
```

Meta-Variables

Dealing with meta-variables in Prolog is also not difficult. If these meta-variables appear in the theory file, this will be because they are contained in a fact. When facts are translated normally, the initial letters of each of each argument is translated from upper-case (indicating a constant in KIF) to lower-case (indicating a constant in Prolog). If a fact contains a meta-variable, we need to ensure that it is not translated into lower-case, but remains upper-case. This means that Prolog will interpret it as a variable. In action rules, the meta-variables also remain as variables, with a class declaration declaring the class that this meta-variable must take.

6.5 Updating the Ontology

There are two situations in which the KIF ontology must be altered. One is when mismatches are diagnosed and the KIF ontology must be refined accordingly; this is dealt with by the refinement system. However, it is also necessary to keep the KIF ontology up to date with respect to the effects of actions that have been performed. This affects only the facts, and not any other ontological objects.

As has been discussed previously, the KIF ontology is not updated every time an action is performed; instead, the PA's Prolog ontology is updated and a record kept of the changes made. However, as soon as planning stops, whether this is because of execution failure or because the goal has been achieved, it is essential that the KIF ontology is updated as well. This is performed by the updating system.

This process is fairly simple. First, the facts to be updated need to be translated from Prolog representation (e.g., *money(agentX,100)*) to KIF representation (e.g., *(Money AgentX 100)*). The appropriate place to insert the fact is searched for — in this case, as part of the declaration of the individual *AgentX* — and the fact is included. If the

effect of an action is to negate a fact, then this fact is searched for in the appropriate place and removed.

6.6 Summary

In this chapter we have described the three non-central subsystems of ORS, whose interaction is described in Chapter 4.

Chapter 7

Results and Evaluation

7.1 Aims of Evaluation

The purpose of this chapter is to provide evidence for the hypothesis we outlined in the introduction:

> Using dynamic ontology refinement to locate and correct ontological mismatches between agents can enable successful communication which would otherwise be impossible.

We outlined three aims of the project that would fulfil this hypothesis:

1. To provide a framework in which agents with first-order, largely similar ontologies can diagnose ontological mismatches between them;
2. To integrate this framework into a system, ORS, that enables an environment where planning agents can use this ability to reach goals that would otherwise have been unreachable. This system must be fully automated;
3. To evaluate these abilities against genuine examples of ontological mismatches, to demonstrate that these abilities are useful and can be successfully performed.

Our approach to this is twofold: firstly, we explore the importance and relevance of the theory described in Chapter 5; secondly we demonstrate what ORS is capable of doing, and place this capability in the context of real-world ontologies.

The evaluation of the system is important because this demonstrates the applicability of the theory. However, the scope of the theory is much broader than the scope of the system, since the system, inevitably having to be placed in a particular context, is a reflection of the theory as implemented in that context only. The system reflects the implementation of the theory within a planning context, but the theory is not inextricably linked to a planning context and could be implemented in many other contexts. We therefore explore firstly how important and useful the theory is within the context of the system, by evaluating the performance of the system, and then investigate the broader implications of the theory, and discuss its usefulness in other domains.

In evaluating the system, we wish to establish answers to the following questions:

- How broad are the capabilities of the system?
- How well do these capabilities correspond with external examples of ontology mismatch; *i.e.*, does the system fix problems that actually exist?
- In what areas does the system fail to perform satisfactorily, and why does this occur? Which of these are limitations of the system, that could be overcome by further implementation, and which are limitations of the theory?

In evaluating the theory, we wish to answer these questions:

- How could the theory be used to create a version of the system which worked in different domains?
- What areas of ontological mismatch is the theory inapplicable to?
- How much further could the theory be developed to extend beyond its original context?

7.1.1 The Context of the Solution

Both the theory and the system have been developed within particular contexts, as discussed above. By context, we mean not a particular domain, but the restrictions they are subject to: for example, the theory is only directly applicable for ontologies written in restricted KIF, and would need to be adapted and extended to work in the context of a different representation. The purpose of this chapter is to evaluate both how well the theory and the system perform within those contexts, and how they could be extended to other contexts. In order to make the evaluation clear, we explicitly state the contexts of both the theory and the system.

7.1.1.1 Context of the Theory

The theory has been developed to deal with first-order ontologies, and is not directly applicable to other forms of ontological representations, such as Description Logics. The presence of precondition refinement in the theory introduces some concept of rules, and thus change, which occurs through these rules. The ideas behind the theory may well be applicable to a much broader domain; this is discussed in Chapter 8. Since first-order logic is more expressive than most ontological representations, we believe developing the theory in this context gives it greater flexibility than if we had developed it for a more restricted representation. However, the particular theory of refinements discussed in Chapter 5 is restricted to the context of first-order logic, and is not directly applicable to other forms of ontological representation.

7.1.1.2 Context of the System

The system is built on top of the theory, and thus experiences all the restraints of the theory. That is, the system is designed to deal with first-order ontologies. In addition, in order to make the development of a system viable, further restraints have been

added. Firstly, a particular first-order ontological representation, namely KIF, has been chosen, and, in fact, deals only with a subset of KIF (see Section 5.5). Secondly, the context for the detection of ontological mismatch, and the evaluation of whether a particular refinement was successful, has been chosen as planning. This does not mean that the theory is only applicable to a planning domain, or, indeed, that it is only applicable to KIF ontologies; this is simply the context that has been chosen for the implementation of the system. The rationale behind this choice is discussed in Section 4.1.3.

7.1.2 Evaluation Issues

There are some difficult issues that need to be considered in the evaluation. These are mostly concerned with the fact that this kind of technology is new: ontologies are developed and altered without the assumptions that the techniques provided by our system are possible; thus the techniques used are not geared towards this. Additionally, the background in which this kind of technology would be most useful, such as the Semantic Web, are themselves only partially developed. It is important to develop the techniques that will be essential for their smooth running, such as the ability for agents of different background to communicate, but in the developmental stages there is simply not enough background information for a thorough evaluation of how these techniques would really perform on the Semantic Web. Evaluation must instead be performed on approximations of what the technology will be like, and estimations of how good these approximations are. As the technology develops, the evaluation will improve, and the systems can be developed accordingly.

7.1.2.1 Finding Suitable Ontologies

The ideal ontologies for evaluating our system would be:

- ones for which we had several different versions, so we could find genuine ontological mismatches;
- ones which described some kind of planning environment;
- ones which are written in the appropriate simplified from of KIF.

Unfortunately, such ontologies proved impossible to find. There are few ontologies for which different versions are publicly available; it is far more common to make only the most recent version of an ontology available. It is very difficult to find suitable ontologies in the planning domain. Historically, there has been a separation between the ontology community and the planning community; this undesirable separation has recently been addressed in a workshop at ICAPS 2005 (International Conference on Automated Planning and Scheduling) [McNeill et al., 2005]. The result of this is that, although ontologies are important in planning, they have tended to be regarded merely as means for testing the efficiency of planners and planning techniques. There has therefore been little incentive to document changes made to planning ontologies, and to keep records of old versions.

There are numerous planning ontologies, usually written in PDDL, for which genuine different versions are not available. There are also some large ontologies, not designed for a planning environment, and written in a variety of different versions, which are publicly available.

Our solution has been to take what ontologies we can for which we have different versions, rewrite a sub-section of them in restricted KIF, and impose a planning scenario on top of them by adding action-rules. In some cases, the mismatches that occurred were found in ontological objects that do not have a direct equivalent in restricted KIF; these were rewritten as restricted-KIF objects. Slightly different ontologies were created for different agents, that reflected the ontological mismatches we discovered between the different versions of the ontology. Additionally, we developed planning

scenarios in KIF representations, and introduced plausible ontological mismatches between agents. These ontologies were run on the system to determine whether the system was capable of appropriately diagnosing and refining them.

7.1.2.2 Methods of Altering Ontologies

Another evaluation challenge is the way in which ontologies are currently updated. Since the techniques implemented in this project are not currently available to ontology editors, it is assumed that matching and updating of ontologies will not be done automatically, but by human users. Thus, when ontologies are altered, no effort is made to describe these alterations in ways that could be understood by automated agents. Generally, explanations about how new or altered objects fit into the ontology are given in natural language commenting, assuming that any user will be able to interpret them. Most agents are not able to interpret natural language comments; an agent's vocabulary consists only in the vocabulary of the ontology, and thus in order to be able to interpret how unknown objects fit into the ontology, they need to be able to relate them to existing ontological objects. This does not mean that our automated techniques do not work on existing ontologies available in different versions; it is often possible to disregard commenting and deduce the ontological difference purely by reference to existing ontological objects. However, we believe that the fact that assumptions are made that humans will interpret these changes means that it is much harder for an automated system to successfully interpret them. For example, a human may change the name of an object because it occurs to them that the name is not fully descriptive. A human reader would be able to understand that this name was related to the old name, and why it had been changed. However, this information would not be accessible to an agent. It is not necessary that these updates should conform to some standard semantic mapping formalism, or that ontology changes should be limited to specific prescribed changes. But if the ontology updater was assuming that agents

might require this information, there would less tendency to rely on natural language interpretation for describing changes, rather than explaining new ontological objects in terms of old or existing ones, not changing names in such a way that the connection is only obvious to those that can interpret natural language, and so on.

Although our system performs reasonably even on existing ontologies, we claim that the success rate of ontology matching would be much higher if people updated ontologies with an assumption that these changes might be investigated automatically. Thus the results of the evaluation, although they show the system to be successful in many cases, do not reflect the true potential of the system.

7.2 Real World Ontologies

We have based our evaluation on three real-world ontologies, which are available online, and for which different versions can be found: AKT, PSL and SUMO. These ontologies are all written in first-order logic, which is essential if they are to be used to evaluate ORS. Naturally, none are written in the restricted KIF we have used for ORS, and thus some amount of manual translation has been necessary in order to run them on ORS. However, the fact that they are first-order ensures that the mismatches we discover in them are relevant to ORS.

These three ontologies were chosen because, of the limited amount of ontologies for which it is possible to access several different versions, these were the most compatible with our restricted KIF.

7.2.1 AKT

Advanced Knowledge Technology (AKT) [AKT, 2002] is a collaborative project involving the Universities of Aberdeen, Edinburgh, Sheffield, Southampton and the

Open University. The project aims to develop and extend a range of technologies providing integrated methods and services for the capture, modelling, publishing, re-use and management of knowledge. The project aims to facilitate knowledge use by addressing six bottlenecks in the engineering and management of knowledge:

1. Acquiring Knowledge
2. Modelling Knowledge
3. Reusing Knowledge
4. Retrieving Knowledge
5. Publishing Knowledge
6. Maintaining Knowledge

There are many different sub-ontologies in AKT; we have examined two of these: AKT-portal and AKT-support. These further divide into several smaller ontologies:

- AKT-portal: docs; events; load; new; organizations; projects; research-areas; techs.
- AKT-support: additional-stuff; basic; foundations; frames; load; new; time.

The AKT ontology was developed primarily for the CS AKTive Portal application, which geographically visualises information in order to provide researchers with a view of the computer science landscape in the UK.

These ontologies are available in several versions: AKT-portal in versions 1, 2.1, 2.2, 2.3 and 2.5; AKT-support in versions 1, 2.1, 2.2 and 2.5. Although many of these ontologies are very small, they nevertheless provide between them many mismatch

examples; far more than are provided by PSL and SUMO. The mismatch examples chosen for the practical evaluations are from the AKT-portal-organizations ontology.

The AKT ontology is written in KIF, and so would seem to be rather a suitable ontology to test ORS on. However, the main way in which we restrict the KIF that ORS uses is by limiting the complexity of class definitions, and complex class definitions form the heart of the AKT ontology. It is thus unsuitable as input for ORS both because ORS cannot handle such class definitions and because the AKT ontology is very static: there are no actions and few relations and individuals. In creating the AKT ontology to run on ORS, we have had to rewrite the ontology to a fairly large extent.

7.2.2 SUMO

Suggested Upper Merged Ontology (SUMO) [SUMO, 2005] was developed within the Standard Upper Ontology Working Group, and its purpose is to promote data interoperability, information search and retrieval, automated inferencing and natural language processing. SUMO is not specific to a particular domain; instead, it provides a structure and a set of general concepts upon which domain ontologies could be constructed.

The SUMO website has access to various past versions of the ontology. We have investigated mismatches that appear between versions 1.50 and 1.51, and 1.51 and 1.52.

The SUMO ontology is written in SUO-KIF (Standard Upper Ontology Knowledge Interchange Format) [SUO-KIF, 2003], which was derived from KIF to support the definition of the Standard Upper Merged Ontology [SUMO, 2005]. It is thus related to standard KIF. Again, we have had to impose a planning context onto the SUMO ontology and translate various terms so that they comply both with restricted KIF and with our planning scenario.

7.2.3 PSL

Process Specification Language (PSL) [PSL, 2005] defines a neutral representation for manufacturing processes. Its goal is to create a process representation that is common to all manufacturing applications, generic enough to be decoupled from any given application, and robust enough to be able to represent the necessary process information for any given application. This representation would facilitate communication between the various applications because they would all speak the same language.

PSL is also written in KIF, and again differs from the required input to ORS because it is static and depends on complex class definitions. We have thus had to impose a planning context onto it and translate it where necessary into restricted KIF.

Less information about the differences between versions of the ontologies was available for PSL than for AKT and SUMO. All that was available was a commentary on the changes that had been made between versions 2.0 and 2.1.

7.3 Practical Evaluation

We have evaluated ORS against the three real world ontologies described above: PSL, SUMO and AKT. We have also evaluated the system against three ontologies that are inspired by genuine planning ontologies, for which we have invented plausible mismatches. These ontologies are from the PLANET (European Network of Excellence) repository [PLANET, 2004]. In all cases, ORS successfully diagnosed and refined a range of ontological mismatches, resulting in the successful execution of plans which had initially been inexecutable.

It should be noted that the three real-world ontologies each had a large number of ontological mismatches between their different versions, for which only a few were

chosen for the practical evaluation. The statistics produced in the theoretical evaluation represent all of the mismatches found in these ontologies; however, not all of them have been used in example scenarios and run on the system. This is because attempting to tie these mismatches together within a plausible planning scenario that can be overlaid onto the ontology is difficult. Plausible goals must be developed, together with action-rules that would take ontological objects as preconditions and effects, so that these mismatches could be discovered within the context of the system. Because this was a slow and difficult task, it was not possible to encode a large number of these mismatches in each ontology. The mismatches were chosen so as to illustrate a range of different ontological mismatches ORS can perform, and to be mismatches that ORS was capable of diagnosing and refining. We have not ignored the ontological mismatches that ORS could not perform; these are discussed in detail in Section 7.4, where we provide statistical evaluation of what proportion of mismatches ORS can refine, and a discussion concerning those that it cannot. The practical evaluation was intended to demonstrate that those mismatches that we claim ORS can perform are indeed successfully performed. There seemed no point in demonstrating that ORS cannot successfully refine mismatches that we do not claim that it is able to.

7.3.1 Results

In this section, we describe the ontologies that we have evaluated the system with, and explain which refinements were performed in the successful execution of the plan for each. Brief output of the system can be found in Appendix C. More detailed output, together with full ontologies before and after refinement and descriptions, with differences highlighted in lines, of how the ontologies changed during refinement can be found on the project website[1].

For each ontology, we give the goal to be achieved and explain what it means. We then

[1] `http://dream.inf.ed.ac.uk/projects/dor/`

give the plan that was produced, explaining briefly the significance. Each plan step in the plan output by Metric-FF and interpreted into Prolog has a list of arguments that cover every individual mentioned in the preconditions and effects of the relevant action rule. The order of the arguments is determined by the order in which they are declared in the PDDL domain file, which is determined by the order in which the translation process encounters them. Since it is not obvious what these individuals refer to just from reading the plan, without reference to the ontology, we will briefly explain them where relevant. The action names for each plan step are shown in bold.

We then indicate at which of these plan steps execution failed, and explain the diagnosis given by the system. After each refinement has been implemented, a new plan is produced, which is given. This plan is sometimes identical to the first plan. This does not mean that no change has taken place: the underlying ontology has been altered, allowing the agent to respond appropriately to questions about the preconditions. However, this does not always require a different plan to be produced: the same plan can be executed successfully with the updated ontology. Sometimes the plan steps are the same but the number of arguments they list is different, due to changes in the preconditions and effects; sometimes plan steps must be added or removed from plans. Additionally, plan steps that have been successfully executed before failure occurred will not appear in the new plan unless they need to be repeated, since replanning will not occur in the same environment as the original planning: the environment has been changed through the actions performed thus far.

We will explicitly state the preconditions and effects of plan steps where this is useful for understanding the mismatch.

In each example, the following information is given:

- A list of the refinements that are performed;

- A list of all the pertinent stages, including the goal, all of the plans, the point of failure and the diagnosed refinement;
- An explanation of what occurred, particularly focused on why the particular refinements were made.

In each case, the edited output of the system is displayed in a Refinement Output figure, and an explanation of this output is given in the main body of the text. Note that the output is given in Prolog notation, rather than in KIF notation as is standard in the rest of the thesis. This is because it is taken directly from output from ORS, which is in Prolog.

7.3.1.1 Online Shopping

This ontology, and the mismatches it encodes, were created by us, based on a plausible scenario for a planning agent in a Semantic Web. The relevant output is shown in Refinement Output 1.

Refinements performed:

1. tightening class restrictions
2. incorrect agent contacted
3. propositional anti-abstraction

The PA, named `shoppingAgent`, has a task to buy the book `ourMutualFriend`. The PA produces a plan (Plan 1) to achieve this. Note that the slightly incongruous argument `aiGroup` is present because the PA believes it is necessary to belong to an online group in order to buy items, and this is an online group that it belongs to.

Refinement Output 1 Online Shopping Ontology

- **Goal:**

  `has(shoppingAgent,ourMutualFriend)`

- **Plan 1:**

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

- **Failure:** at action `putInBasket`. There was a surprising question: `class(aiGroup,shoppingGroup)`. `aiGroup` is currently constrained to be of class `group`, due to the definition of `registered-member`, which is a precondition of `put-in-basket`.

- **Refinement: tightening class restrictions**

- **Plan 2:**

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

- **Failure:** at action `joinGroup`, immediately after requesting the action to be performed.

- **Refinement: incorrect agent contacted** - the agent replied, on questioning, that it was not able to perform the appropriate action, and the information that that agent could perform that task was thus removed from the meta-ontology.

- **Plan 3:**

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

- **Failure:** at action `buy`. There was a surprising question: `money(shoppingAgent,dollars,Amount)`, which has a predicate match with expected precondition `money(shoppingAgent,100)`. An extra argument of class `currency` needs to be added as the second argument.

- **Refinement: propositional anti-abstraction**

- **Plan 4:**

  `[buy(ourMutualFriend,dollars,shoppingAgent)]`

- **Success**

This plan fails at the first action: `putInBasket`. Diagnosis reveals that the class restrictions on one of the arguments was too loose: it was restricted to class `group`, and thus the planner allowed the argument to be instantiated to an object with class `academicGroup`, since this is a subclass of `group`. In fact, the class restriction should have been `shoppingGroup`. The necessary refinement is performed.

The PA then replans to produce Plan 2. This again fails at the first action, which this time is `joinGroup`. Diagnosis reveals that the incorrect agent was contacted. The necessary refinement is performed.

The PA replans to produce Plan 3. This plan is identical to Plan 2, because the previous refinement affected the meta-ontology. Thus the plan produced, from the top level ontology, is the same, but the PA's ability to execute the plan, which depends also on the meta-ontology, is improved, because this time it knows to contact a different service-providing agent. This time failure occurs at the third action, `buy`. Diagnosis reveals that the precondition `money(shoppingAgent,100)` was expected to be `money(shoppingAgent,dollars,_56691)`, thus indicating that the `money` predicate is missing an argument of class `currency`.

The PA replans to produce Plan 4. This is successfully executed.

7.3.1.2 Blocks World

This ontology is based on a standard planning scenario, which has been adapted as necessary. The relevant output is shown in Refinement Output 2.

Refinements performed:

1. Switching arguments
2. locating an incorrect fact: removing an incorrect fact from the ontology and altering the effects of a rule

Refinement Output 2 Blocks World Ontology

- **Goal:**

  `correctOrder(blockX,blockY,blockZ)`

- **Plan 1:**

  `[pickUpFromBlock(blockZ,armOne,blockX),`

  `pickUpFromTable(armOne,blockY),`

  `moveToBlock(armOne,blockX,blockY),`

  `moveToBlock(armOne,blockY,blockZ),`

  `order(blockZ,blockY,blockX)]`

- **Failure:** at action `moveToBlock(armOne,blockX,blockY)`. There was a surprising question: `holding(armOne,blockX)`, which has a predicate match with expected precondition `holding(blockX,armOne)`; the arguments are transposed.

- **Refinement: Switch arguments**

- **Plan 2:**

  `[pickUpFromBlock(blockZ,blockX,armOne),`

  `moveToBlock(armOne,blockX,blockY),`

  `moveToBlock(armOne,blockY,blockZ),`

  `order(blockZ,blockY,blockX)]`

- **Failure:** at action `order`, immediately after the request to perform this action was made.

- **Refinement: incorrect fact located** - The service-providing agent confirmed it could provide the service; thus an incorrect precondition was searched for. `empty(armOne)` was found to be incorrect, and it was believed to be true as it is a precondition of a previous action. Thus the incorrect fact was removed from the ontology and the effects of the relevant rule altered.

- **Plan:**

  `[emptyArm(armOne),order(blockZ,blockY,blockX)]`

- **Success**

The goal is to place the three blocks in the correct order, which is a single column in alphabetical order, with the arm empty. The PA produces Plan 1 to achieve this. Note that the last action of this plan, `order`, is an action which verifies that the order of the blocks is correct and that the arm is empty. This fails at action `moveToBlockarmOne,blockX, blockY)`. Diagnosis reveals that an unexpected query was received: `holding(armOne, blockX)`. This corresponds with an expected precondition: `holding(blockX,armOne)`. They have the same arity and classes of arguments, but these classes are reversed. Switching arguments is therefore diagnosed.

The PA replans to produce Plan 2. This fails at the action `order`. This failure occurred immediately after a request was made to perform `order`. A problem precondition was discovered: `empty(armOne)`. The Shapiro algorithm reveals that this fact was an effect of the action `moveToBlock`. An assumption has been made that the arm can only hold one block; thus when it puts that block down, it is now empty. However, in reality, the arm can hold more than one block, and so is not necessarily empty after a block has been put down. The agent that performed `moveToBlock` is consulted to see if it believes that `empty(armOne)` is an effect of the action and, since it agrees with the agent that attempted to perform `order` that it is not true, PA removes it as an effect of the action. It must also remove the fact that states this is true, since this has been shown to be incorrect. Note the assumption entails that this is also falsely believed to be an effect of the action `moveToTable`. However, since this action is not part of the plan, this is not noticed. This will be fixed if it causes plan execution failure at another time.

The PA replans to produce Plan 3. This is successfully executed.

7.3.1.3 Lift Scheduling

This ontology is based on a standard planning scenario. The relevant output is shown in Refinement Output 3. Note that `floor` has been abbreviated to `fl` and `person` has

been abbreviated to pers.

Refinements performed:

1. locating an incorrect fact
2. negating a precondition
3. propositional abstraction

The PA produces Plan 1. This fails at action serveLiftCustomer(persOne,flOne, liftOne,securityCleared,flZero,onsiteEngineer). The plan failed immediately after a request was made to perform serveLiftCustomer(persOne,flOne, liftOne,securityCleared,flZero,onsiteEngineer) Diagnosis shows that there is a problem precondition: inService(liftOne,onsiteEngineer). The Shapiro algorithm reveals this was not made true by a previous action, but was a fact in the original ontology. Refinement thus removes this fact from the ontology.

The PA replans to produce Plan 2. Failure occurs at the action serveLiftCustomer(persOne,flOne,liftOne,securityCleared,flZero,onsiteEngineer). The plan fails after a query about hasAccess(persOne,flOne). This was an expected query, and it was believed that a negative answer was appropriate, because a precondition is not(hasAccess(persOne,flOne). Since this answer was not appropriate, we must negate the precondition, and replace it with hasAccess(persOne,flOne).

The PA replans to produce Plan 3. This fails at action serveLiftCustomer(persOne, flOne,liftOne,securityCleared,flZero,onsiteEngineer). Diagnosis reveals that a query was received about call(persOne,liftOne), which was not expected. This matches expected precondition call(persOne,liftOne,securityCleared), but has a lower arity.

The PA replans to produce Plan 4. This is successfully executed.

Refinement Output 3 Lift Scheduling Ontology

- **Goal:** and(served(persOne),served(persTwo),served(persThree))
- **Plan 1:** [*serveLiftCustomer*(persOne,flOne,liftOne,securityCleared,flZero,onsiteEngineer), *serveLiftCustomer*(persTwo,flZero,liftOne,securityCleared,flOne,onsiteEngineer), *serveLiftCustomer*(persThree,flTwo,liftOne,securityCleared,flZero,onsiteEngineer)]
- **Failure:** at action serveLiftCustomer(persOne,flOne,liftOne, securityCleared,flZero,onsiteEngineer), immediately after the request to perform the action was made.
- **Refinement: incorrect fact located.** The service-providing agent it could perform the task. An incorrect preocndition inService(lifeOne,onsiteEngineer) was discovered to be a fact in the original ontology, and removed.
- **Plan 2:** [*moveEmptyLift*(flOne,liftOne,flZero),*serviceLift*(liftOne), *serveLiftCustomer*(persOne,flOne,liftOne,securityCleared,flZero,onsiteEngineer), *serveLiftCustomer*(persTwo,flZero,liftOne,securityCleared,flOne,onsiteEngineer), *serveLiftCustomer*(persThree,flTwo,liftOne,securityCleared,flZero,onsiteEngineer)]
- **Failure:** at action serveLiftCustomer(persOne,flOne,liftOne, securityCleared,flZero,onsiteEngineer), immediately after the expected question hasAccess(personOne,floorOne): the expectations about the truth value of this precondition must thus be incorrect.
- **Refinement: negate precondition**
- **Plan 3:** [*grantAccess*(flOne,persOne),*grantAccess*(flZero,persTwo), *grantAccess*(flTwo,persThree), *serveLiftCustomer*(persOne,flOne,liftOne,securityCleared,flZero,onsiteEngineer), *serveLiftCustomer*(persTwo,flZero,liftOne,securityCleared,flOne,onsiteEngineer), *serveLiftCustomer*(persThree,flTwo,liftOne,securityCleared,flZero,onsiteEngineer)]
- **Failure:** at action serveLiftCustomer(persOne,flOne,liftOne, securityCleared,flZero,onsiteEngineer). There was a surprising question: call(personOne,liftOne), which has a predicate match with expected precondition call(personOne,lifeOne,securityCleared). The final argument must thus be removed.
- **Refinement: propositional abstraction.**
- **Plan 4:** [*serveLiftCustomer*(persOne,flOne,liftOne,flZero,onsiteEngineer), *serveLiftCustomer*(persTwo,flZero,liftOne,flOne,onsiteEngineer), *serveLiftCustomer*(persThree,flTwo,liftOne,flZero,onsiteEngineer)]
- **Success**

7.3.1.4 AKT

The relevant output is shown in Refinement Output 4.

Refinements performed:

1. precondition anti-abstraction
2. propositional anti-abstraction
3. adding a class

Note that the mismatch diagnosed as precondition anti-abstraction corresponds in the original ontology to a condition that is added to a class. Since that cannot be reproduced in restricted KIF, it is interpreted as adding a precondition to an action. Additionally, the propositional anti-abstraction is not a genuine mismatch from the AKT ontology, but is included as a device to reveal the new class, which is a genuine mismatch.

This goal states that *researchAgent*, which is an agent of type *researcher* should be enrolled as a member of *refinementSig*, which is of class *Sig* (Special Interest Group), which is itself of class *Organisational-Unit*. The PA forms Plan 1 to achieve the goal. The plan contains only one action: to enrol the agent in the Sig. The arguments refer to the organisation to be joined, the researcher that wishes to enrol, the institute to which the researcher belongs, the subscription fee that is payable, and the general research interest which must belong to both the organisation and the researcher. The fact that there is only one action indicates that the preconditions of this rule are already believed to be fulfilled. This plan fails at action `becomeMemberOrganization`. The communication with the service-providing agent indicates the existence of a precondition `hasEmailAddress(researchAgent,_297427)`. This precondition indicates that

Refinement Output 4 AKT Ontology

- **Goal:**

  `memberOrganization(researchAgent,refinementSig)`

- **Plan 1:**

  `[becomeMemberOrganization(refinementSig,researchAgent, schoolOfInformaticsAtEdinburgh,subscriptionFee, ontologyRefinement)]`

- **Failure:** at action `becomeMemberOrganization`. There was a surprising question: `hasEmailAddress(researchAgent,Address)`, which does not match any expected preconditions. It is therefore assumed to be a missing precondition.

- **Refinement: Precondition anti-abstraction**

- **Plan 2:**

  `[generateEmailAddress(researchAgent, schoolOfInformaticsAtEdinburgh), becomeMemberOrganization( refinementSig,researchAgent, schoolOfInformaticsAtEdinburgh, subscriptionFee, ontologyRefinement)]`

- **Failure:** at action `becomeMemberOrganization`. There was a surprising question: `memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh)`, which has a predicate match with expected precondtion `memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh)`. The expected precondition is missing a third argument of class `organizationUnit`.

- **Refinement: propositional anti-abstraction and adding a class**

- **Plan 3:**

  `[becomeMemberOrganization(refinementSig,researchAgent, ukCsDepts,schoolOfInformaticsAtEdinburgh,subscriptionFee, ontologyRefinement)]`

in order for an agent to join the Sig, it must have an email address. The second argument is a Prolog indication of an uninstantiated variable. This means that the service-providing agent wishes the PA to instantiate this variable. It might be that any instantiation is acceptable; the service-providing agent just wishes to ensure that it is true for some argument; this is the case in this situation. In some situations, only certain instantiations would constitute an acceptable answer. The PA attempts to link this precondition to any of its existing preconditions, but, in this situation, fails. Therefore it concludes that this is a missing precondition, and refines its ontology accordingly. A warning message is given to indicate that this may not be correct: it may be that the precondition is linked to an existing precondition in a way the PA cannot determine.

The PA replans to produce Plan 2. This fails at action `becomeMemberOrganization`. Diagnosis reveals that a query was received about `memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh,ukCsDepts)`, which was not expected. However, this matched expected precondition `memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh)`. The class of this extra argument was not known, but the service-providing agent informed the PA it was of class `educationalOrganizationalUnit`. This class does not exist in PA's class hierarchy, so it needed to ask the superclass of this class and fit it into its class hierarchy accordingly.

The PA replans to produce Plan 3, which is successfully executed. Note that in order to form an appropriate plan, the newly refined precondition `memberAcademicUnit(Agent,Unit,MetaVar)` must be instantiated. Since PA did not know of the class `educationalOrganizationalUnit` before the refinement, it does not know any instances of the class. However, it has encountered an instance during querying: this was the prompt for the refinement. Therefore it adds the object `ukCsDepts` to its ontology as an individual of class `educationalOrganizationalUnit`. This is the only value `MetaVar` can be instantiated to.

7.3.1.5 SUMO

The relevant output is shown in Refinement Output 5.

Refinements performed:

1. domain abstraction
2. predicate abstraction

This goal indicates that a given agent wishes to know a particular communication. The PA produces Plan 1. This fails at the first action, `doB`. A query was received about `resultRel(actionA,objectA)`, which was not expected. This matched the expected precondition `resultRel(actionA,entityA)`, except that the class of the second argument was different: an expected argument of class `entity` mismatched an actual argument of class `object`. `object` is a superclass of `entity`, so domain abstraction was diagnosed.

The PA replans to produce Plan 2. This plan is the same as the previous plan, except `entityA` has been replaced by `objectA`. The plan failed at action `tell`. A query was received about `represents(physicalA,objectA)`, which was not expected. No precondition matched the name, and the PA could not find any related precondition, but the service-providing agent said that the precondition `refers(physicalA,objectA)` is a subclass of `represents`, so predicate abstraction was diagnosed.

The PA replaned to produce Plan 3. This was successfully executed. Note that `represents(physicalA,objectA)` is an effect of the action `doB`; in order to make `refers(physicalA,objectA)` true, `doA` must be performed instead.

7.3.1.6 PSL

The relevant output is shown in Refinement Output 6.

Refinement Output 5 SUMO Ontology

- **Goal:**

  knows(agentB,communicationA)

- **Plan 1:**

  [*doB*(entityA,physicalA,domainA,actionA,agentA),

  tell(communicationA,agentB,physicalA,entityA)]

- **Failure:** at action doB. There was a surprising question: resultRel(actionA,objectA), which has a predicate match with resultRel(actionA,entityA). The class of objectA (object) is a subclass of the class of entityA (entity).

- **Refinement: domain abstraction**

- **Plan 2:**

  [*doB*(objectA,physicalA,domainA,actionA,agentA),

  tell(communicationA,agentB,physicalA,objectA)]

- **Failure:** at action tell. There was a surprising question: represents(physicalA,objectA). The predicate name is a superclass of the predicate name of an expected precondition: refers(physicalA,objectA).

- **Refinement: predicate abstraction**

- **Plan 3:**

  [*doA*(objectA,physicalA,communicationA,actionA,agentA),

  tell(communicationA,agentB ,physicalA,objectA)]

- **Success**

Refinements performed:

1. propositional anti-abstraction
2. predicate anti-abstraction

The goal is to make occurrence `occ` into a simple object. The PA produced Plan 1, which failed at action `ruleOne`. A query was received about `mono(subOccTwo,occOne, actTreeOne)`, which was not expected. This matches the precondition `mono(subOccOne,subOccTwo)`, but they have different arity. Thus propositional anti-abstraction was performed.

The PA replaned to produce Plan 2. This failed at action `ruleThree`. A query was received about `sameTree(occTwo,occ)`, which was not expected. No preconditions matched the name, but the PA found its precondition `sameGrove(occTwo,occ)` was a superclass of `sameTree`. Predicate anti-abstraction was diagnosed.

The PA replaned to produce Plan 3. This was successfully executed.

7.3.2 Missing Refinements

The refinements performed using the three real-world ontologies show a good range of the abstractions and anti-abstractions. However, there are some types of refinements that do not appear naturally in these ontologies. These are refinements such as precondition refinement and discovering problem preconditions and using the Shapiro algorithm to discover their source. This is because these refinements are concerned with action rules and individuals. Such ontological objects are central to planning ontologies but are not so important to static ontologies. Usually, these have few, or no, individuals and action rules. It is thus difficult to demonstrate these refinements using our off-the-shelf ontologies; these are demonstrated using our planning ontologies.

Refinement Output 6 PSL Ontology

- **Goal:**

  `simple(occ)`

- **Plan 1:**

  `[ruleOne(occTwo,occOne,subOccTwo,subOccOne,actTreeOne),`
  `ruleOne(occOne,occTwo,subOccOne,subOccTwo,actTreeOne),`
  `ruleTwo(occTwo,occOne),`
  `ruleThree(occ,occTwo,occOne) ]`

- **Failure:** at action `ruleOne`. There was a surprising question: `mono(subOccTwo,occOne,actTreeOne)`, which has a predicate match with expected precondition `mono(subOccTwo,occOne)`. The expected precondition is missing a third argument of class `activityTree`.

- **Refinement: propositional anti-abstraction**

- **Plan 2:**

  `[ruleOne(occTwo,occOne,actTreeOne,subOccTwo,subOccOne,`
  `actTreeOne),`
  `ruleOne(occOne ,occTwo,actTreeOne,subOccOne,subOccTwo,`
  `actTreeOne),`
  `ruleTwo(occTwo,occOne),`
  `ruleThree(occ,occTwo,occOne)]`

- **Failure:** at action `ruleThree`. There was a surprising question: `sameTree(occTwo,occ)`. The predicate name is a subclass of the predicate name of an expected precondtion: `sameGrove(occTwo,occ)`.

- **Refinement: predicate anti-abstraction**

- **Plan:**

  `[ruleFour(occ,occOne),`
  `ruleFour(occ,occTwo),`
  `ruleThree(occ,occTwo,occOne)]`

- **Success**

7.3.2.1 Mismatches that Cannot be Diagnosed by ORS

Precondition Abstraction

One of the mismatches we have defined is not amenable to discovery through plan execution failure. Mismatches are only highlighted by plan execution failure if they can cause plan execution failure; some mismatches cannot. Precondition abstraction entails an action rule having tighter restrictions on it than necessary. It may lead to a situation where it appears impossible to form a plan, when the correct preconditions would allow the formation of a plan. However, if a plan is formed, it cannot fail due to this mismatch, as all the correct preconditions - and more - are fulfilled. This situation is inherent in any version of ORS that diagnosed mismatches purely through plan execution failure. If the primary aim of the system is to enable successful plan execution, then this does not constitute a problem. However, it does mean that an ontology cannot be automatically updated to cover this change using ORS.

Domain Anti-abstraction

The system also does not diagnose domain anti-abstraction. This differs from the inability to diagnose precondition abstraction because the latter inability is a limitation of the system, albeit an unavoidable one in the context, whereas the former reflects the fact that domain anti-abstraction does not reveal an underlying mismatch in the ontologies.

Consider, for example, the situation in which PA has an expected precondition:

```
(Holding Arm-A Block-A),
```

It then receives the following query from the service-providing agent:

```
(Holding Object-A Block-A)
```

Clearly, there is an instantiation clash here; the precondition, which concerns an object named `Arm-A`, does not match the query, which concerns an object named `Object-A`. PA knows that `Arm-A` is of class `Arm`, and so there seems to be a class mismatch, as `Object-A` is of class `Object`. However, PA also knows that class `Arm` is a subclass of `Object`. Thus, although `Object-A` is of a class not anticipated by PA, it does not represent a ontology mismatch.

7.4 Evaluation of the Theory

We have evaluated above how good the system is at performing the tasks that we designed it to perform. In this section, we attempt to evaluate how well this ability meets the actual demands of ontological mismatch. We evaluate this by examining the mismatches we encounter in the three off-the-shelf ontologies for which we have different versions. We have explained in the beginning of the section why evaluating ORS against such ontologies is not especially favourable to it; this is explained in more detail in Section 7.4.1, where we explore the situations in which ORS would be unable to perform the appropriate refinement.

We define four different categories into which mismatches may fall, one of which is further subdivided:

1. ORS could refine the mismatch;
2. ORS could not currently refine the mismatch, but non-major changes to the system would allow ORS to refine it;
3. ORS could not refine the mismatch. This is because:
   (a) ORS did not have sufficient functionality;

(b) This particular mismatch is outside the scope of the project;

(c) This mismatch is irrelevant to an automated system — this is usually a change to commenting or formatting;

(d) This mismatch could not occur in the restricted KIF that ORS is designed to use;

(e) This mismatch could not be highlighted in a planning context.

4. The information we had about the mismatch was insufficient to diagnose which of the above categories it would fall into.

From the above categories, it is clearly desirable that as many as possible fall into category 1; such mismatches indicate a successful outcome for ORS. However, given the assumptions we have had to make in ORS and the unsuitability of the ontologies against which it is evaluated, it is to be expected that many will fall into categories 3b)- 3e). A poor outcome for ORS would be represented by many mismatches falling into category 3a). It may seem fairly arbitrary whether mismatches are assigned to category 3a) or category 3b): is a mismatch unrefinable because it is outside the scope of the project, or because ORS does not have sufficient functionality? A generous interpretation may say that anything that ORS has not been designed to perform is outside the scope of the project, and therefore nothing should be assigned to 3a); a less generous interpretation may say that any mismatch that cannot be refined for reasons that are not due to restriction of context (3c-3e) indicate a lack of functionality in ORS, and therefore nothing should be assigned to 3b). We take a middle line here. We assign mismatches to 3b) if they belong to categories that we have discussed in previous chapters and explained the importance to the project of not to attempt to include such functionality in ORS; otherwise we assign them to 3a). Under such circumstances, mismatches being assigned to 3a) is the worst outcome; nevertheless, a large number of mismatches being assigned to 3b) would indicate that ORS is not especially well

adapted to the task it is attempting to perform. A large number of mismatches in 3d) does not directly provide a poor outcome for ORS, since we have made and justified a decision to limit the ontologies with which ORS can operate, and also the general problem is undecidable. However, a high percentage in this category would indicate that the amount of work required to adapt ORS to full KIF or to another ontological representation would be reasonably large.

It may also seem arbitrary whether mismatches fall into category 2 or category 3a). We only assign mismatches to category 2 if they could be diagnosed and appropriately refined by techniques that are similar to the techniques already contained in the system and would not require much extension of the theory. For each of the kinds of mismatches that we assign to this category, we give a justification of why we believe it meets this criterion.

Given that it is hard to get full information about these ontologies, we must expect a few mismatches to fall into category 4: this is not a reflection on ORS, merely on the data we have available. In most cases, this allocation is given to mismatches that could be easily refinable in certain circumstances, but would not be in others, and where the context is not clear enough to determine which of these is the case.

It is important to stress that ORS has not actually performed all those mismatches that fall into category 1: testing ontologies on ORS is a complicated process, involving rewriting the ontologies in restricted KIF, overlaying a planning context in such a way that the mismatches can be highlighted, developing agents to perform the tasks that the ontology requires and writing ontologies for them that are subtly and appropriately different, and so on. It was only practical to run ORS with a subsection of these mismatches, chosen to fully indicate the functionality of the system. The remainder of the mismatches were evaluated theoretically; we assign them to category 1 if they correspond with mismatches that we have demonstrated ORS can refine.

In some cases, there were mismatches that fell into more than one category. For ex-

ample, in the AKT ontology, a common mismatch was the addition of a new class. Adding a new class can be performed by ORS, but ORS deals with only simple class definitions, and thus if there is complex information attached to this class definition, it cannot be added. We considered these to be two different mismatches: the adding of the class itself, stating its place in the class hierarchy, which would fall into category 1, and the adding of complex information about that class, which would fall into category 3d).

Below, we summarise our results in tables and piecharts, where the proportions of mismatches that fall into each category are illustrated. The significance of these results is explained in Section 7.4.1. The complete ontologies from which these results came, and the complete set of their mismatches, can be found through links from the project website[2]. The mismatches on the website are highlighted to illustrate which category each mismatch falls into.

In Table 7.1, we give the total number of mismatches in each category for each ontology and in total. Table 7.2 then restates this information as percentages of the total number of mismatches. Tables 7.3 and 7.4 give different perspectives on these percentages. In Table 7.3, we omit categories 3c) and 4. The number of alterations to commenting is really not relevant to an evaluation of the performance of ORS: to an automated system commenting can be disregarded. Additionally, the evidence provided by mismatches of category 4 is impossible to evaluate, as we do not know what they indicate about the performance of the system. We thus consider that this table gives the percentages for each category of significant mismatches. In Table 7.4, we also omit categories 3d) and 3e), which are those mismatches that are outside the context of the project. These percentages therefore reflect the proportion of relevant mismatches in each category. We then illustrate this information in pie-charts.

Note that these results are much more heavily influenced by the AKT ontology than by

[2] http://dream.inf.ed.ac.uk/projects/dor/

Category	PSL	SUMO	AKT	Total
1	15	16	95	126
2	6	0	15	21
3a)	0	2	0	2
3b)	5	16	8	29
3c)	3	17	17	37
3d)	0	2	83	85
3e)	1	6	10	17
4	6	1	1	8
Total	36	60	229	325

Table 7.1: Numbers of Mismatches per Category

the other two, since we have far more data from the AKT ontology. The effect of this is discussed in Section 7.4.1.

7.4.1 Analysis of Results

The results illustrated in the tables and charts indicate a reasonably good performance for ORS. ORS can perform 38.8% of all mismatches, 45.0% of significant mismatches, and 70.8% of relevant mismatches. Here, we discuss the kinds of refinements that fall into each of the categories.

- **Category 1**

  The practical evaluation discussed in Section 7.3 illustrates that a wide range of the potential mismatches we identified can actually be found in these ontologies. Additionally, we see that a fairly high proportion of mismatches fall into this category, despite the unsuitability of the ontologies, discussed in Section 7.1.2. However, we should qualify these results to some extent by explaining in more

Category	PSL	SUMO	AKT	Total
1	41.7	26.7	41.5	38.8
2	16.7	0	6.6	6.5
3a)	0	3.3	0	0.6
3b)	13.9	26.7	3.5	8.9
3c)	8.3	28.3	7.4	11.4
3d)	0	3.3	36.2	26.2
3e)	2.8	10.0	4.4	5.2
4	16.7	1.7	0.4	2.5

Table 7.2: Percentage of Mismatches per Category

Category	PSL	SUMO	AKT	Total
1	55.6	38.0	45.0	45.0
2	22.2	0	7.1	7.5
3a)	0	4.7	0	0.7
3b)	18.5	38.1	3.8	10.4
3c)				
3d)	0	4.8	39.3	30.3
3e)	3.7	14.3	4.7	6.1
4				

Table 7.3: Percentage of Mismatches per Significant Category

Category	PSL	SUMO	AKT	Total
1	57.7	44.4	80.5	70.8
2	23.1	0	12.7	11.8
3a)	0	5.6	0	1.1
3b)	19.2	44.4	6.8	16.3
3c)				
3d)				
3e)				
4				

Table 7.4: Percentage of Mismatches per Relevant Category

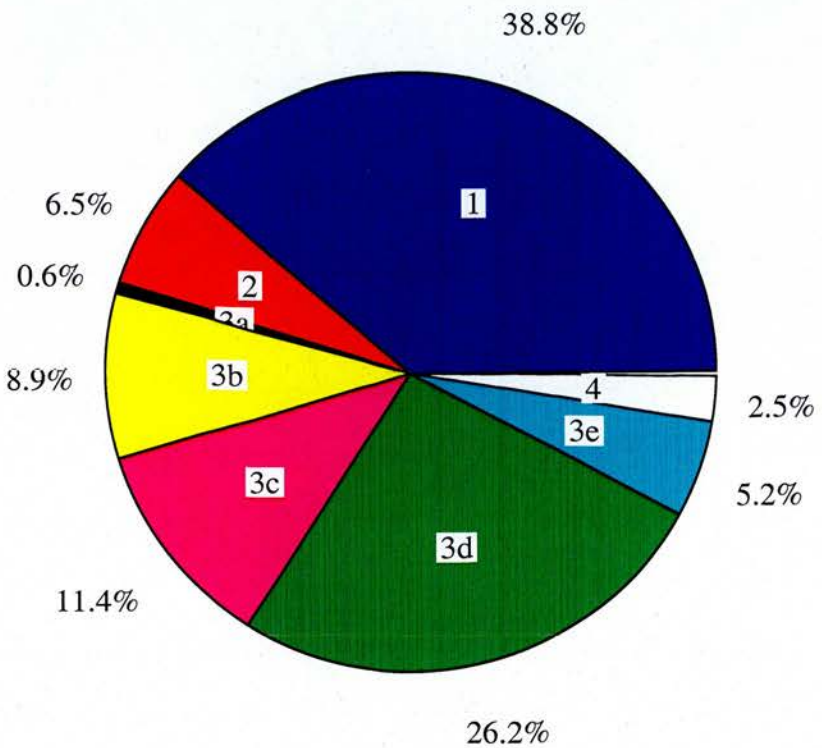

Figure 7.1: Total Percentage of Mismatches per Category

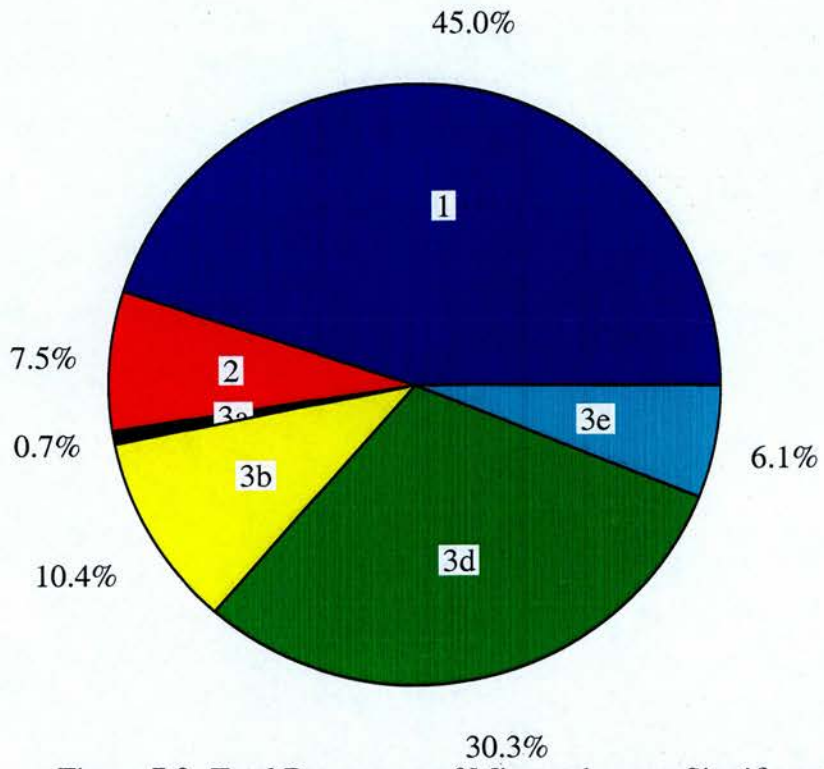

Figure 7.2: Total Percentage of Mismatches per Significant Category

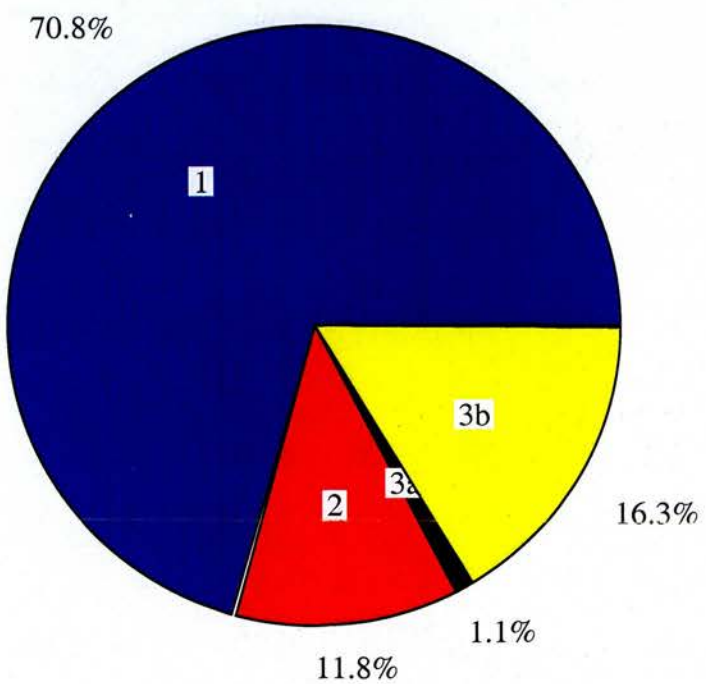

Figure 7.3: Total Percentage of Mismatches per Relevant Category

detail what the percentages mean. The results obtained from the AKT ontology have a high impact on these good results; this is especially so for the results for relevant categories. Whilst we can find a reasonable spread of mismatches in AKT, this result is heavily influenced by some of the more simple refinements, and in particular, adding classes. A high percentage of category 1 mismatches from AKT involve merely adding a class to the ontology; thus, although this is a good result for ORS because it can perform this, it does not indicate that the full functionality of ORS is not being used quite as widely as might be imagined from the results.

- **Category 2**

  Only 6.5% of mismatches fell into this catagory. All of these fell into three different groups:

  - Changing the superclass of a class:

    ORS can already alter the class hierarchy in simple ways: for example, by adding a class. It cannot, however, alter the superclass of an existing class. This mismatch would be easy to refine, but would be difficult in most circumstances to diagnose.

  - Changing the class of an instance:

    This is very similar to the above situation.

  - Adding relations and functions to the ontology:

    In some situations, it would be easy to diagnose this: if, for example, this new relation or function was a precondition of an action, it would be revealed through questioning. It would not be difficult to refine the ontology to add a new relation if the name of the relation and the number and classes of the arguments was known. It would, however, be very difficult to determine whether an unknown predicate represented a relation or a function.

In cases of uncertainty, it would probably be better to add a relation rather than a function, as this is a more general case.

- **Category 3a)**

  The results in this category are very low, which is a good result for ORS. In fact, only two of all the mismatches fell into this category. These two occurrences both appeared in the SUMO ontology.

  - The first was concerned with an axiom: the original axiom contained a double implication as the top-level operator, and this was altered to a single implication. This could be viewed as removing an axiom, though it is represented as an alteration of an existing axiom. It is not clear how this problem could be identified by ORS, though the rule could be marked as unusable if it failed to be executed successfully when no problem precondition could be found.
  - The second concerned an object called `NullSet` which was defined as an individual in the original ontology but redefined as a class in the updated ontology. Its original definition was that it was *the* set that contains no instances, whereas the updated definition was that it was *any* set or class that contains no instances; hence the change of categorisation. Such a change would be difficult for ORS to diagnose.

- **Category 3b)**

  The percentage of mismatches that fall into this category are far more numerous than those that fall into the previous category. The percentages are still much less than those for category 1, illustrating that the limitation of scope that we enforced on this project does not impact the effectiveness of ORS to a very large degree. Nevertheless, nearly 9% of all mismatches, and over 16% of relevant mismatches, fell into this somewhat undesirable category. We briefly examine those that did, justifying why they were not put into category 3a), and discussing

why ORS could not be expected to refine them.

We consider these mismatches to be outside the scope of the project because they fall into one of two categories:

- Multiple interacting mismatches:

  Often, more than one mismatch is made simultaneously in such a way that these mismatches are dependent on one another. ORS can diagnose two mismatches if this can be done consecutively: first one mismatch is diagnosed and refined, then plan execution failure occurs again, then the second mismatch is diagnosed and refined. This is clearly not possible if the mismatches affect one another.

  Many mismatches that fall into this category are those that involve the rules. Adding a new rule is complex. It is impossible to diagnose through plan execution failure (thus should perhaps be assigned to category 3e)), and it also involves several different ontological objects: all the preconditions and effects; and hence can be viewed a series of multiple interaction mismatches. Equally difficult, and fairly common, is the replacement of existing rules by one or more restated rules. In this situation, it might be possible to diagnose that the old rule no longer existed; it would not be possible to diagnose how it should be replaced.

  Sometimes, single preconditions are restated as several new preconditions, or vice versa, or in some other way preconditions are restated. For example, in one situation,

```
(=>
      (true ?FORMULA1 True)
      (true ?FORMULA2 False)))
```

  is replaced by:

```
(not (and (true ?FORMULA1 True)
```

```
(true ?FORMULA2 True))))
```

These two statements are logically equivalent. However, for an automated system to detect the complex interaction of mismatches here could be difficult. This could only really be done as a human does it: through analysis of the interaction of logical operators. This would be easily detectable for a system that could interpret logic, but not for a system that was attempting to diagnose mismatch only with reference to existing ontological objects, and without a built in understanding of logical equivalence.

We believe that this is the situation that most suffers from the assumption that human users, rather than automated users will be the ones to interpret and update ontologies. If automated users are considered, it will probably become much less common to restate conditions in this way. However, the problem of adding rules remains: it would not be desirable to completely remove the potential to add rules to an ontology, and yet these will be difficult to refine, particularly in a planning context.

- Arbitrary changes to the ontology:

  Again, we believe that this occurs more frequently because of the assumption that the process of updating ontologies is not automated. One way in which this manifests itself is the correction of errors in the original ontology. This would fall into the category of arbitrary changes to the ontology, since the original error in the ontology would be random. We have made the assumption that the ontologies we are working with do not contain errors, and therefore we consider these problems to be outside the scope of the project.

  Changing the names of ontological objects is also very difficult to diagnose. ORS can deal with this in a situation where it is done systematically: for example, changing the name to a sub- or super-class. However, if it is done arbitrarily - perhaps, for example, because it sounds better or makes

more sense - this is hard to detect automatically. One example of this is a name change from `R&D-INSTITUTE` to `R-and-D-INSTITUTE`. There are ways in which this problem could be lessened: for example, an online dictionary could be used to detect synonyms, and standard conversions between natural language words and symbols (such as `&` and `and`) could be listed. However, we have not attempted to implement any of these, as we do not consider arbitrary changes to be within the scope of the project.

- **Category 3c)**

  Mismatches that fall into this category are not really interesting to the evaluation. However, the fact that they occur reasonably frequently (11.4% of all mismatches) indicates that there is much information in these ontologies which is not readable automatically by agents that cannot interpret natural language, and that some effort is put into updating this information. This may not be important: in some situations, commenting merely explains what is clearly stated by the ontological objects themselves, but in a more human-understandable manner. However, it may be that the commenting explains complex interactions of ontological objects that could not be derived by an automated system interpreting only the ontological objects themselves. This is particularly true in refinement situations: the relation of the new information to the old is explained in a way that might not be explained explicitly using the ontological objects themselves. Again, this is a reflection of the assumption that this information is relevant only to human users. If automated systems perform this task more commonly, effort will be put into explaining these changes using ontological objects rather than through commenting.

- **Category 3d)**

  Mismatches that could not occur in the restricted KIF that ORS uses occurred very rarely in the PSL and SUMO ontologies, but were frequent in the AKT

ontology (over 25% of all mismatches). This may seem surprising, since AKT is written in KIF, and we might therefore expect it to correspond well with the mismatches ORS is able to perform. However, the complex class definitions of AKT mean that there is actually a significant difference in the AKT ontology and ORS acceptable ontologies. Almost all these mismatches involved adding to or altering slots on the classes. The only exceptions to this are two cases of classes being assigned an additional superclass; in our restricted KIF, we only allow each class to have a single superclass.

The only example of this outside the AKT ontology was a single occurrence in the SUMO ontology. This involved quantification, which we do not allow in our restricted KIF.

- **Category 3e)**

  Mismatches in this category are not very common: just over 5% of all mismatches. They are exclusively concerned with objects being removed from the ontology. It is possible in a planning context to diagnose that certain ontological objects should not be used in certain situations, but it is not possible to detect that they should be removed from the ontology altogether. This could potentially lead to problematic situations. Consider the situation in which a certain instantiation of a relation is believed to be a precondition of a rule. If this relation is removed from the ontology, it will not be possible to make a particular instantiation of it true. Any rule for which it was an effect would have been altered so that it was no longer an effect when the relation was removed. It would be possible to use our techniques to determine that it was falsely believed to be an effect for each of the rules for which this was previously believed to be the case. However, it would not be possible to determine that it was erroneously believed to be a precondition: thus we would be in a situation where it was not possible to fulfil the (supposed) preconditions of an action, and hence it may not be possible to form a valid plan. This is a problem that is inherent in the approach we have taken to

ontology refinement: discovering ontological mismatch through plan execution failure.

- **Category 4**

  There are few mismatches in this category: only 2.5% of the total number of mismatches. These mostly occur in the PSL ontology, for which we do not have a great deal of information about the ontology. We do not have access to both versions of the ontology for PSL, only to a description of what changes have been made, and thus it is not always possible to determine how these match our refinements. These mismatches concern statements such as *The definition of* `subtree_embed` *has been tightened.* Without knowing what the definition of `subtree_embed` is, or exactly how it has been tightened, it is not possible to tell if this is something ORS could refine or not.

7.5 Comparison with Similar Work

An important part of the evaluation is showing not only that the system works and that it performs useful tasks, but also placing the work in the context of other work in the field.

Chapter 3 describe the fields that also deal with the problems of mismatch and refinement. In this section, we examine in more detail how our work relates to those fields.

7.5.1 Ontology Mismatch

The field in which we are working in this project is that of ontology mismatch. As has been described in Section 3.1, this is a field in which much work has been done, particularly in recent years.

The techniques and systems described in Section 3.1 vary considerably, and in this section we attempt to explain how our work relates to these various approaches.

There are a few main variables that can be seen in this field:

1. How much of the ontology is available when the mapping/merging/alignment process is carried out? Is the whole of both (or all) of the relevant ontologies available, or are there some inaccessible parts?

2. How highly structured is the ontology, and how much of this structure is considered during mapping? Is the mapping/merging/alignment process concerned only with the concepts (classes) and concept hierarchy of the ontology, or are more complex ontological objects such as relations considered?

3. How similar are the ontologies assumed to be?

4. When is this mapping process to take place?

5. How much of the ontologies are mapped?

6. How much user interaction is required?

For ORS, the answers to these questions are:

1. We assume that only one of the ontologies is fully visible whilst this process is taking place; very little is known about the other ontology;

2. We consider highly structured ontologies. Mapping concepts is a part of what ORS is capable of, but it is also capable of mapping much more complex ontological objects: relations, functions and axioms;

3. We assume that the ontologies are already reasonably similar;

4. The mapping process takes place incrementally, whilst interaction is taking place;

5. Only those parts of the ontologies that are directly involved in communication failure are mapped;

6. The process is fully automated.

These questions reveal the main difference between our work and that of most of the work in the field (for example, [Giunchiglia and Shvaiko, 2003, Giunchiglia et al., 2005a, Doan et al., 2003, Campbell and Shapiro, 1998, Kalfoglou and Schorlemmer, 2003a, Wiesman et al., 2002]). For a more standard ontology mapping system, the answers to the above questions would be:

1. Both (all) of the ontologies are assumed to be fully visible;

2. The process is entirely concerned with mapping concepts;

3. The ontologies do not need to be similar;

4. The mapping process takes place before interaction is commenced;

5. All of the ontologies are mapped;

6. The process is interactive.

The answers to these questions underline the difference in emphasis between our work and more standard work on ontology mapping. Our work is both stronger and weaker: it is effective even when there is very little information about what we are mapping to (we have explained in previous chapters why we believe this to be important), but, unlike standard ontology mapping, it can only work with ontologies that are already fairly similar. Rather than being another version of ontology mapping, our work is, in fact, orthogonal to standard ontology mapping. Much of ontology mapping could be said to be concerned with the following questions:

"given that we have a number of pre-defined ontologies that agents have access to, how can we map between these ontologies so that agents that subscribe to different pre-defined ontologies can communicate. "

Our work, on the other hand, attempts to answer the question:

"given that the pre-defined ontologies that agents have access to are not, in fact, static but are developing and are being altered by individual users, how can we refine mismatches that are found between these two similar but non-identical ontologies?".

Thus our approach to ontology refinement is complementary to the standard approach to ontology mapping. We still require ontology mapping to bridge the gap between ontologies that are significantly different. We add to the field of ontology mapping by removing the implicit assumption that pre-defined ontologies are fixed and do not have different versions. Our approach is necessarily rather different, because it is reasonable to assume that pre-defined ontologies are available for analysis, and this does not have to be done at run-time. However, this is not a reasonable assumptions for small changes and updates that have been made by any number of people, and thus this must be done during run-time. We therefore have constraints on us that are not present in standard ontology mapping: user interaction is not practical; we cannot assume we can access very much information about the other ontology; the PA cannot understand terms in other agent's ontologies unless it can relate them to terms in its own.

The relationship between our work and standard ontology mapping is illustrated in Figure 7.4.

Ontologies A1 - A3 represent versions of Ontology A that have been altered slightly, as B1 and B2 do for Ontology B. Thus for an agent using Ontology A (with no variations) to communicate with an agent Ontology B, ontology mapping would be required; for an agent using Ontology A1 to communicate with an agent using Ontology A2, dynamic ontology refinement would be needed; for an agent using Ontology A1

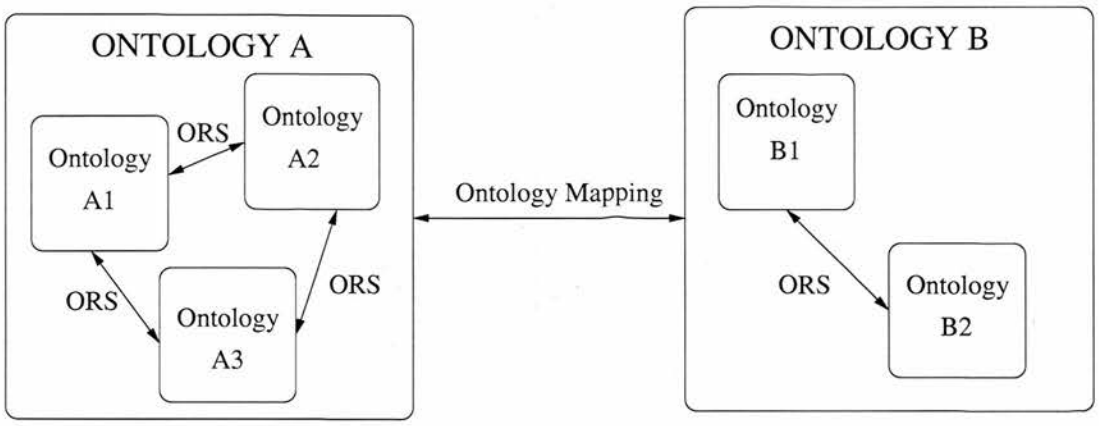

Figure 7.4: Ontology Mapping and ORS

to communicate with an agent using Ontology B1, both would be needed. In the latter case, a complete mapping between Ontology A and Ontology B would be established before any communication took place, and then, during runtime, dynamic ontology refinement could be used to deal with the instances where the ontologies of the agents do not actually conform to expectations, due to the variations between Ontology A and Ontology A1 and between Ontology B and Ontology B1.

Note that we do not claim that ORS can, as it stands, be used in tandem with any particular ontology mapping system to perform such tasks; we are merely trying to illustrate how the theories are related.

7.5.1.1 Determining Potential Mismatches

In terms of developing techniques to identify the causes of mismatches, our approach of predefining the potential mismatch can be seen in other systems. These were outlined in Section 3.1.5. Many of these are different to those that we have defined, because the way in which they define their ontologies differ. For example, many of Visser's potential mismatches [Visser et al., 1997] are based on the notion of ontological objects being defined by a tuple $< T, D, C >$, where T is the term that is being defined, D is what it is defined in terms of, and C is the ontology-concept description

to be defined, in natural language. Since this is not how objects in our ontologies, nor in most ontologies, are defined, the potential mismatches defined on this basis are not relevant.

Many of the mismatches defined are, in fact, classes of mismatches rather than particular mismatches: for example, those defined by Wiederhold [Wiederhold, 1997] and Shaw and Gaines [Shaw and Gaines, 1989]. Additionally, almost all the ideas examined were not implemented, but are merely theoretical description of how things might change.

7.5.2 Belief Revision

The theory of belief revision is discussed in detail in Section 3.2. In this section, we discuss its relevance to ORS. The central difference with the theory of ORS is that belief revision is not concerned with representation issues but with the problem of whether a particular fact is true or false in a particular situation. In belief revision it is assumed that the signature is the same for all instantiated facts. It is thus not directly relevant to those parts of the ORS concerned with signature change.

In addition to signature change, ORS is also concerned with some aspects of theory revision: namely, changing action rules (including by precondition refinement) and removing facts from the ontology when they are discovered to be problem preconditions. The first of these issues is again not particularly relevant to standard belief revision, because refinements that alter rules are concerned with the structure of those rules, and with the question of how the preconditions or effects of a rule can be changed so that the rule is more correct. Even if we are facing a situation where we know that a rule is incorrect, but do not know how we can alter it appropriately, we do not need to make use of belief revision, because all we can do is mark this rule unusable. The difficult issues of belief revision revolve around what effect removing a belief from a set of be-

liefs has on that set, and how consistency can be maintained. However, action rules are not interdependent in the way that beliefs in a belief set are, so this question is moot. If the rule is marked unusable, then we may not be able to perform that action, but it will not affect our beliefs in the other action rules. It is possible that a more complex version of ORS might have some appreciation of the inter-relation of rules. A very complex ontology might organise rules together into groups of rules that are somehow similar, so that marking one as unusable could affect one's confidence in other similar rules. However, no such complex interaction between rules exists in ORS, nor is it seen as a priority to introduce such notions.

The second of these issues, that of removing facts from the ontology, is more directly relevant to belief revision. ORS currently takes a naive approach to this by simply removing the offending fact. However, belief revision could be used to find a more sophisticated approach to this problem. This could become relevant if agents negotiated over which was to change its ontology, rather than the PA always taking this upon itself (see Section 8.4.2). There is, however, a significant difference in the aims of belief revision and the aims of the ORS. Belief revision is concerned only with maintaining a consistent knowledge base. ORS, on the other hand, not only has to maintain a consistent knowledge base (or theory of the ontology), but also attempts to make this knowledge base consistent with respect to the knowledge of other agents. Some kind of belief revision could prove useful as an aspect of the negotiation process. An agent's willingness to remove a fact from its ontology on the recommendation of another agent could depend on what the epistemic entrenchment of that belief is, and how integral that belief is to the ontology: how many other facts would be affected if it were removed. At the moment, there is no interdependency of beliefs within the ontology; they are all considered to be independent entities. However, if a more complex view of beliefs were incorporated into the ontology, then this would need to be managed using some kind of belief revision.

Whilst it seems that belief revision could be of some use in a more advanced version of ORS, it is clear that the theory behind ORS is of an essentially different nature to that of belief revision.

7.5.3 Theory Change

Some of the ideas discussed in Section 3.3 provide an interesting comparison to our work. The work done on the development of mathematical theories by Lakatos [Lakatos, 1976] and Hayes-Roth [Hayes-Roth, 1983] is concerned with altering theories in the light of counter-examples. For example, this may be by specifically excluding the counter-example from the theory, by generalising the theory so that the counter-example no longer refutes it, or perhaps by abandoning the theory altogether. These attempts to alter theories have some relevance to the need in ORS to alter rules after their failure. Lakatos's *surrender* (abandoning a conjecture in light of counter-examples) is similar to what is done in ORS when a rule fails in a way that defeats diagnosis: it is simply marked as unusable. Some of the ways in which ORS alters preconditions, such as pre-condition anti-abstraction, could be viewed as instances of Lakatos's *monster-barring* (modifying a definition to exclude counter-examples), Lakatos's *lemma incorporation* (adding a precondition) and Hayes-Roth's *exclusion* (barring the theory from applying to the current situation). Hayes-Roth's *retraction* (restricting a theory's predications to be consistent with observations) is similar to ORS altering the effects of an action after it has been identified by the Shapiro algorithm as a cause of failure. One way in which ORS might be improved is to increase its ability to diagnose and refine faulty rules, in which case closer investigation of these techniques might be useful in considering more complex ways in which rules might alter. However, not all of the techniques would apply, because both Lakatos and Hayes-Roth are considering the more general case of theory change rather than specifically action-rule change. In addition, these approaches do not deal with the problem of signature change.

The machine learning approaches described in Section 3.3.3 are rather different to the approach taken by ORS due to the different aims of the environment in which they are learning. The emphasis is generally on learning over a large set of training examples, and thus they are not directly relevant to the situation we are investigating, where changes must be immediately implemented when failure is encountered. The situation would be more similar to that of standard machine learning if we kept some kind of past history 8.4.1 of changes that had been implemented before, but nevertheless the approach taken to using these examples to derive useful information about a current situation would necessarily be rather different to that of standard machine learning techniques.

The explanation-based learning approach is more similar to our approach, because EBL machines can create justified generalisations from single training instances [Ellman, 1989]. They also tend to depend, to some extent, on background knowledge of the domain under study, just as ORS depends on a reasonable, though not exact, understanding of mismatched ontologies. The main difference between EBL machines and ORS is that EBL machines, like many methods of theory change, is concerned with how *explanations* can be altered to converge to some better explanation; essentially this is equivalent to investigating how rules can be altered to better reflect how services are performed. This is certainly quite close to some aspects of ORS, but again does not address the issues of signature refinement. As with Lakatos's and Hayes-Roth's work, if ORS's approach to fixing faulty rules was made more sophisticated, then EBL techniques should be explored more carefully.

Some of the approaches to correcting faulty formulae described in Section 3.3.4 are again similar to some aspects of ORS. For example, the idea of adding a condition so that a faulty formula becomes reliable in a more restricted situation is similar to precondition anti-abstraction. However, the focus is again on how to alter theories to be correct, and does not address the signature alteration part of ORS. The work on

program debugging and repair, described in Section 3.3.5, is again different in emphasis. The aim is to create a program that is correct rather than to deal with matching two different representations. The point is to determine what the overall effect of a program is, rather than on whether specific objects within the program, such as predicates, are correctly represented. Thus it is loosely relevant to those changes in ORS that alter rules, but not to those that alter the signature. Additionally, the environment in which these operations take place, namely the programming language of choice, mean that the demands and assumptions are rather different to those that can be made for a first-order ontology.

7.5.4 Fault-Catalogue and Model-Based Diagnosis

The work on diagnosis in these fields constitutes an interesting comparison to our own diagnostic work. However, there are some key differences between them that make the majority of work in fault-catalogue and model-diagnosis inapplicable in our situation. A key difference is that the finding of faults is far more focused in ORS; the agent communication gives information that considerably narrows down the search and, in many cases, immediately identifies the offending ontological object. There is therefore no need to compile conflict sets that list all the possible combinations of faults, which is the key mechanism of diagnosis in these fields. However, in ORS we are not allowing the full possibility of multiple faults. We are not using what would be termed in these fields as the single fault assumption (*i.e.*, that the problem is caused by a single fault) because we allow the possibility of the problem being caused by multiple faults that can be resolved serially, but we do rule out the possibility of the problem being caused by multiple faults that interact with one another. We intend, at a later stage, to incorporate this ability into the system. This would significantly complicate the diagnostic process and creating conflict sets could then become a useful approach. It seems likely that it would be rather difficult to calculate these conflict sets, especially if no limit was placed

on the number of ontological objects that could be interacting. Another difference is that these fields are concerned with internal inconsistency in a system, whereas we are interested in differences between two distinct representations.

There is much more focus on how to find the causes of the problem than on how to fix them, which is often left for the user to determine [Scholbach and Cornet, 2003]. However, in ORS, the finding of the offending ontological object and the determining of the way in which it offends is part of the same process. In a few cases it is only possible to determine with what object the fault lies, and not how to resolve it, but in such cases the user is not involved and the fault is 'repaired' by removing the object from the ontology. Additionally, much of this work is concerned with physical systems, and the components that are considered as potential problems are physical components. Thus the narrowing down of the error requires experimentation on the part of the user. In our system, this narrowing down is done partially through observation of past interaction and reasoning about ontological differences by the PA, and partly through further interaction with relevant agents. Hence, although some kind of focusing process is necessary in both approaches, the mechanisms are rather different. In general, the diagnosis in these fields tends to be interactive, and the automation is focused on making it easier for a user to discover and fix problems. However, because ORS is concerned with agent interaction, it is essential that both the diagnosis and the repair of faults is fully automated; the agents must be able to decide for themselves how to proceed when faults and mismatches are encountered.

It seems initially that this project more closely resembles fault-catalogue diagnosis, because diagnosis is based on a list of potential mismatches. However, the assumption that the ontologies of the agents are the same, and thus mismatches need only be searched for if problems unexpectedly occur, means that the ontology of the PA can be considered to be a model of the ontologies of the other agents, and the problem of determining how the ontologies are mismatched is then equivalent to the problem

of determining how the 'system' (the ontology of the other agent) differs from the 'model' (the PA's ontology). The predefined potential mismatches can then be seen as a way in which the diagnostic processes is guided through the possible mismatches between the model and the reality.

7.6 Summary

We believe that ORS successfully performs refinements that are useful in a multi-agent system where agents do not always have identical ontologies. In this section, we have demonstrated the grounds for such a belief, and given statistics that illustrate the performance of ORS against real ontology mismatches. These results indicate that ORS is already capable of performing well in the domain for which it is designed and, following suggestions in this chapter and the following one, could be extended to perform even more successfully. The results of this chapter indicate that the techniques embodied by ORS could be the foundation of a system that can perform important tasks in a real world situation, for which there are currently significant gaps: namely, allowing agents with non-identical ontologies to interact successfully in a large and complex multi-agent system, such as the Semantic Web.

We believe that we have thereby fulfilled the three aims of the project, outlined at the beginning of this chapter and in Chapter 1, by providing a framework for agents to diagnose ontological mismatches, integrating this within a fully automated system, and evaluating the usefulness of this against real-world examples of ontology mismatch. We have thus demonstrated the validity of our hypothesis: that using dynamic ontology refinement to locate and correct ontological mismatches between agents can enable successful communication which would otherwise be impossible.

Chapter 8

Further Work

The large and complex field in which we are basing our research could generate several further projects as extensions to the work we have done thus far. Not only has our research solved an interesting subset of the problem at hand, it has also raised many ideas about how this could be applicable to a wider field. As has been discussed in previous chapters, we had to limit the scope of what could have been an open-ended project in order to produce valid results in the timescale. This means that there are many avenues for further work: relaxing these constraints and exploring how the work could be developed in less constrained domains.

An obvious way for the system to be developed would be in ways that would make it more immediately usable on the Semantic Web. The system is currently not suitable for use on the Semantic Web, partly due to these limitations of scope, and partly due to the development being focused on first-order logic rather than on a Description Logic based representation, such as OWL, which is far more commonly used on the Semantic Web.

In this chapter we discuss the main ways in which the system would need to be developed to function on the Semantic Web. In addition, we discuss other ways in which we would like to extend the system beyond its original constraints.

8.1 Adapting to Different Representations

One of the most obvious limitations of the system is that it is restricted to a specific type of ontological representation, one that we cannot expect that any external agents will be using. It would seem sensible to extend the system to make it compatible with full KIF, as this would be a relatively small step from the current system and would make the system usable in a real-world situation. However, if the system is to operate on the Semantic Web, working with full KIF is of limited usefulness; much more useful would be to adapt it to work with OWL. We discuss these two possibilities below.

8.1.1 Extending ORS to Full KIF

This would involve the need to allow for more complex class definitions and to allow for quantification, and would require extensions to the diagnosis techniques. It is not immediately clear how complex details about class definitions could be deduced from the agent interaction. Currently, we only need to find the name of the class and then put a direct question to the agent as to the superclass. Further information may require the revealing of larger sections of the ontology, such as we do not permit in ORS. It is also not obvious how information about quantification could be ascertained through agent querying, unless direct questioning about the quantification of preconditions of actions was permitted. Further research would be necessary to determine the effects on plan execution failure of allowing quantification, and how it could be diagnosed as the cause of plan execution failure.

If we allowed these two aspects of KIF to be included in ORS ontologies, there would also need to be some additional work for the translation system. The class definitions in full KIF are much more complex than those in PDDL, and we would have to consider how these could be represented in PDDL, or whether they could legitimately be ignored in the PDDL files. If we allow quantification over infinite domains, we would

need to devise a method of capturing a finite representation of this in the PDDL files; this would probably involve much loss of expressivity and lead to PDDL files that could not produce plans that would be valid with respect to the KIF ontology. Alternatively, we could use a first-order planner, but this could introduce considerable search problems.

Once the above translation issues have been considered, it would certainly be possible to get ORS to work with ontologies in full KIF, but it is uncertain how good ORS would be at diagnosing some of the mismatches that could occur within these extended parts of the ontology.

8.1.2 Adaptation to OWL

An important way in which the domain of ORS could be extended is to adapt it to different ontological representations. The most obvious representations to choose would be Description Logic based representations such as OWL and the DAML family of languages, since these are very common forms of agent languages and are particularly dominant in current research on the Semantic Web.

Re-implementing ORS so that it works with different ontological representations will require some amount of effort. The central functionality of ORS — diagnosis and refinement — is heavily dependent on the representation of the ontology. Some work will be required to develop diagnostic algorithms that are appropriate for a new representation, and these will be based on analysis of what kinds of mismatches are possible in such a representation, which may differ from the mismatches that are possible in restricted KIF. Additionally, a new translation process will be required to create PDDL and Prolog files from this new representation. The planning system should be unaffected by a change in representation, as long as it is possible to form appropriate PDDL files from the representation. The agent communication system would also be

largely unaffected by such a change, except that the messages passed would be in a new representation.

Nevertheless, although there would be many technical demands in making this change, the theory behind ORS is adaptable to any representation, and the changes would be mostly of an implementational nature. Clearly, the more different to KIF the new representation, the more work would be required to adapt the system. However, OWL has many similarities with KIF, and much of the existing theory would still apply.

8.1.2.1 Applying the Current Refinements to OWL Ontologies

In this section, we concentrate on the adaptation to OWL-DL, rather than OWL-lite or OWL-full. OWL-DL has been designed with a particularly focus on tractability, and is therefore less expressive than a full-first order representation. One of these limitations concerns relations. KIF relations and functions are equivalent to OWL-DL relations; however, OWL-DL relations are always binary, which means that the applications of propositional refinement (altering the arity of a relation or function) would be much more limited. In addition, unary relations, called concepts, can be defined. It would still be possible in theory to turn a concept into a relation and vice versa, though whether this would ever be a useful thing to do is not certain. Since classes of arguments in relations is still important, domain refinement (altering the class of an argument) would be largely unaffected, as would switching arguments. Predicate refinement (altering the class of a relation or function) would also be possible. The implementation of this would be rather different, and rather more natural than the way this is done for KIF ontologies by ORS, since this could be handled by the *subPropertyOf* predicate, which describes hierarchies between relations, rather than by using a meta-ontology. The most difficult refinements to mirror in OWL-DL would be those concerning action rules. As mentioned in Section 2.3.1, OWL-DL is not directly capable of describing services; this is done by OWL-S. However, the way in which services

are described in OWL-S is very different from the inference rule style that we have adopted in KIF. It is not possible to express preconditions and effects with OWL-S, so tags or place-holders are used instead. Thus the diagnostic and refinement rules dealing with incorrect service descriptions would be very different to the current method of dealing with action rules in ORS.

8.2 Improving the Agent Communication System

Currently, we believe that the agent communication system in ORS is sufficiently sophisticated for the needs of ORS. However, if ORS is to be extended and used on the Semantic Web this will cease to be the case, and hence a good preliminary to this process is to improve the agent communication system. The most obvious way to do this is to make the system FIPA compliant. This would also have the desirable effect of bringing ORS more in line with standard agent technology; it would then be able to communicate with any FIPA-compliant agent. Compliance with a standard agent protocol is essential if we wish to use this system for agents not designed by us, and thus is essential for practical application.

Adding such sophistication would not be particularly difficult and would not affect any part of the system except the agent communication system. It might be possible to make use of this additional information in the diagnostic system, since the additional number of performatives provide more information than is provided using more simple performatives: for example, the FIPA performative *refuse* (see Table 8.1) could be used to indicate that the service-providing agent was either incapable of performing that action, or perhaps was unwilling to divulge confidential information. However, deriving useful information from this would be an optional addition to the diagnostic system rather than an essential alteration.

The way in which the existing simple performatives in the ORS agent communica-

tion system correspond to the more complex performatives of FIPA is illustrated in Table 8.1. This shows that a large number of the FIPA performatives do not have any parallel in the ORS performatives, and are not required for the simple agent interaction performed by ORS. This would not be a problem if we were making ORS FIPA-compliant; it could be FIPA-compatible without itself making use of all of the FIPA performatives. The ORS performative *request* is directly comparable with the FIPA performative *request*. However, the ORS performative *question* is compatible with both the FIPA performatives *query-if* and *query-ref*. *Query-if* allows an agent to ask another whether or not some specific statement is true or not, whereas *query-ref* is used by an agent to determine a specific value for an expression. There would thus need to be a distinction made between these two situations in order to make ORS FIPA-compatible. Likewise, the ORS performative *reply* is comparable with several FIPA performatives. These situations would need to be distinguished in order to make ORS FIPA-compatible. All of these situations already occur in ORS and distinguishing between them would not be difficult; in fact, it may add to the clarity of the system.

A very useful addition to the performatives would be to provide a response that indicates a signature mismatch. Currently, the PA will reply *no* to a query if it is not able to instantiate the query appropriately. This might be because its instantiations of that predicate conform to a different signature to the query, or it might simply be that it has no relevant information about that query. For example, if the query is:

(Money PA ?Amount Dollars)

then the service-providing agent is asking the PA to return the amount of dollars it has. The PA will reply *no* to this question if there is a signature mismatch: perhaps the PA has information in the following form:

(Money PA 100), or

(Dollars PA 100).

The PA will also reply *no* if this query matches the signature of service providing agent's ontology, but nevertheless the PA has no appropriate instantiation: perhaps the

	ORS performatives			
	Query			
FIPA performatives	Question	Request	Reply	**No Parallel**
accept-proposal				x
agree			x	
cancel				x
call for proposals				x
confirm			x	
disconfirm			x	
failure			x	
inform			x	
inform-if				x
inform-ref				x
not-understand				x
propagate				x
proxy				x
query-if	x			
query-ref	x			
refuse				x
reject-proposal				
request		x		
request-when				x
request-whenever				x
subscribe				x

Table 8.1: Comparison of FIPA and ORS Performatives

PA's only instantiation is:

(Money PA 100 Sterling).

For refinement purposes, it would be useful to distinguish between these two situations. Possibly the FIPA *not-understand* (see Table 8.1) could be used for this purpose.

We could also create agents that could provide more information about their ontologies: for example, revealing entire action rules. However, for reasons discussed in Section 6.2, we believe that it would be counter-productive to add this functionality. It would make it much easier for ORS to accurately diagnose mismatches, but it would make it unusable in any but purposely designed agent systems and additionally could not be made FIPA compliant, as there are not FIPA performatives that cover this situation.

8.3 Retaining Different Versions of Ontologies

One potentially useful addition to ORS would be the ability to keep several different versions of ontologies: refinements are not performed to a central ontology, but instead copies of the original ontologies are made and refinements are performed on these. There is some justification for such an approach: we cannot always know whether we are correctly diagnosing a mismatch with respect to the ontology of the agent we are interacting with and, even if we are, we cannot be sure that this other agent has an ontology that agrees with that of other agents. Therefore, creating a copy of the ontology to which this refinement is performed and keeping a copy to which it is not performed would allow us to keep our options open as to whether this refinement should be performed or not.

It would be very easy to create different versions of ontologies by making copies of the original ontology every time a refinement was performed. We would need to decide

how far this process should be taken. There are several ways in which this could be done:

1. A new ontology is created every time we perform a refinement, which would result in many ontologies each differing from the original ontology in one specific way;

2. A single copy of the ontology is created, to which all refinements are performed;

3. A new copy of the ontology is created each time a refinement is performed, but this is a copy not of the original ontology, so that the new copy would contain only the most recent refinement, but of the refined ontology, so that the cumulative effects of the refinements could be seen. For example, if the following series of refinements were performed: precondition anti-abstraction, domain abstraction, adding a class, we would afterwards have four ontologies: the original ontology, a copy on which precondition anti-abstraction had been performed, a copy on which precondition anti-abstraction and domain abstraction had been performed, and a copy on which all three of the refinements had been performed. However, this does not seem a very sensible approach, as the order of the refinements is arbitrary;

4. Every time a refinement is performed, a copy of every existing ontology is taken, and the refinement is performed to each of these. Thus there would be a copy of the original ontology to which it was performed, there would also be a copy of every version of the ontology to which only one refinement had been performed to which the new refinement would be performed, and so on. However, this would lead to a combinatorial explosion of the number of ontologies.

5. A new ontology is created for each agent with which the PA interacts. This would have the advantage of disposing of the assumption that there is some more correct ontology to which we can tend towards, thereby implying that our

communication with all agents is likely to improve if we refine our ontology to match a specific agent. Instead, this allows us to assume that agents have different ontologies, and that there is no advantage in trying to create a better ontology, but rather we create many ontologies that allow us to deal appropriately with each agent. Whether we wish to assume that all agents are similar or that all agents are different would depend on the circumstances. If we assume that all agents are different when they are in fact similar, we are creating extra work for the PA, because it will have to repeat the same refinement many times. However, this is not nearly such a severe problem as would be found in option 4;

6. Every time the diagnosis was uncertain, the corresponding refinement could be performed to one version of the ontology but not to another, or if there were a wider choice, each choice could be performed on a given ontology.

The first option would make it easiest to unwind refinements if they later appeared to be inadvisable. However, it would mean that no attention would be given to the cumulative effects of refinements. ORS currently only performs refinements individually; however, by performing many individual refinements, a situation may be reached where an ontological object differs in more than one way from its original version. Such situations could not be reflected in this option. If, for example, propositional anti-abstraction had been performed on a predicate, and then domain abstraction, the PA could chose to use either the original predicate, or the version with an additional argument, or the version with a different class of argument, depending on what was appropriate for the agent with which it was currently communicating. However, it would not be able to use a version of the predicate that had an additional argument and a different class for one of the arguments, since these refinements would have been performed separately.

If the same situation occurred when the second option had been chosen, the PA would be able to use a version of the predicate that had both an additional argument and a

different class for one of the arguments, or use the original version of the predicate to which neither of the refinements had been performed. It would not be able to use a version of the predicate to which only one refinement had been performed.

If this situation occurred in the third option, the PA would be able to use a version of the predicate to which both refinements had been performed, or only the first, or neither. It would not be possible to use a version to which only the second refinement had been performed.

The fourth option provides the most flexible approach for the PA: any version of the predicate that is possible under any combination of the performed refinements would be available. This is, therefore, perhaps the best option. The drawback of this is that it could lead to the creation of vast numbers of ontologies, which would be expensive to store and update. Every time a mismatch was diagnosed, it would need to be performed to several ontologies, which would become increasingly inconvenient as the number of ontologies increased.

The fifth option could provide some element of protection against choosing the incorrect refinement, as the correct refinement would always have been performed on one ontology, and also would keep the number of total ontologies lower than some of the other options. However, this is only an attempt to deal with the problem that diagnosis and refinement are not always certain, and does not deal with the question of whether the PA ought to trust the ontology of the other agent as being more reliable than its own.

Whenever actions are performed, the world state has been changed, and this needs to be reflected in every ontology that exists. After a plan action has been successfully performed, its effects change the facts in the domain. Since these are part of the ontology, the ontology must be updated. Every possible world (all the ontologies) must reflect this way in which the world has changed. This would lead to another disad-

vantage if many versions of the ontology were kept: updating the state would become increasingly onerous.

Another method of retaining the ability to undo undesirable refinements is the method currently employed by the system: simply keeping a list of the refinements that have been performed, so that their inverses can be performed if necessary. This, however, is not a complete solution to the problem. Often, applying refinements to the ontology involves removing information: propositional abstraction involves removing an argument from a predicate definition and all its instantiations; domain abstraction involves changing the class of an argument to a different one, and hence uninstantiating this argument for all occurrences of this predicate, and allowing them to be reinstantiated appropriately. Thus it is not always possible to return to the original ontology simply by performing the inverse refinement; keeping different versions of the ontology provides a more complete solution to the problem. In order to determine which of the above options would be preferable, further research needs to be done into the precise demands of the situation.

One way in which it would be possible to keep the potential for many different versions of the ontology without having to store so much information would be to extend this idea. A more detailed record could be kept of what changes had been made to the ontology, so that it would be easier to rewind it if necessary. Thus, not only would information be saved stating that propositional abstraction had been applied, but also what details of all the information that had been removed from the ontology in applying this refinement.

8.4 Relaxing the Constraints

Another category of changes that could be made to the system are those that are not directly necessary for it to be used on the Semantic Web, but would improve the per-

formance of ORS.

8.4.1 Using Previous History

In the current system, a record is kept of what refinements have been performed. However, this is done only so that the user can observe what changes have been made to the ontology; it is not used by the diagnostic process.

An interesting extension to ORS would be to investigate how it could use this previous history in the diagnosis process. For example, it could be used as supporting evidence when deciding whether to make an uncertain diagnosis. Section 7.3.2.1 discusses how we do not diagnose domain anti-abstraction in ORS, because we don't have enough evidence to suppose that this is a better diagnosis than diagnosing incorrect instantiation with no signature mismatch. However, if we had access to more detailed records of previous mismatches, we could see how many times this particular situation had occurred. If there were several cases where this predicate had caused failure, and in every case the particular instantiation involved a mismatch where the class of one of the arguments was a subclass of the expected class, and this subclass was always the same subclass, we might then suppose that domain anti-abstraction was the more plausible diagnosis.

We could also use this method to accumulate evidence for the other uncertain diagnosis: precondition anti-abstraction. In this case, for reasons discussed in Section 7.3.2.1, we do diagnose precondition anti-abstraction even though we cannot be sure this is correct (this diagnosis is never certain because it may be that there is an unknown connection between the surprising question and one of the preconditions). If information about past occurrences of precondition anti-abstraction was stored, it would be possible to compare the predicate of the surprising question with the predicate in the preconditions. If we had many cases of a particular surprising question always appear-

ing when a particular predicate was a member of the preconditions, and additionally this precondition predicate never appeared in any questions, we may eventually have enough evidence to conclude that some equivalence between these two predicates was likely.

In the current system, these are the only two situations in which we would need to rely on previous history, since the other diagnoses are certain. However, if we were to extend the system to deal with multiple mismatches, this kind of information would become much more important.

Previous history could also be used in detecting anomalous agents; this is discussed in Section 8.4.3.

8.4.2 Negotiating about Refinements

In ORS, there is no negotiation about which agent is to refine its ontology: the PA will always refine its ontology and the service-providing agent will be unaware and unconcerned about the refinements that the PA is making to its ontology. This creates a pattern of refinement which seems to imply that the service-providing agent has an ontology that is somehow objectively better than the ontology of the PA, and that the only desirable way that these two ontologies can be made compatible is through the PA aping the service-providing agent.

We have justified this approach in ORS through appeal to the situation in which refinement is being performed: the PA wants the service-providing agent to perform an action for it, and is thus prepared to do whatever is necessary to ensure this happens, whereas the service-providing agent is unconcerned whether the action is performed or not, and thus will not make any effort to ensure it can be. Moreover, we believe that it is justified to limit the scope of the project, so that it is coherent with the time-scale, and thus we have chosen a simple approach to this problem.

However, in reality most situations in agent-communication systems, and especially those using peer-to-peer architecture, do not conform to this simplistic view of 'knowledgeable' agents and 'ignorant' agents. Rather, each agent is treated as an equal citizen of the agent communication system. It may be that one agent has a more up-to-date or more relevant version of the information than another agent, but this must be established through negotiation between the agents, and cannot be determined merely through examination of their relative positions. If an agent wishes it to be established that it has better information than another agent, it must present evidence for this. The other agent is then not forced to accept the advice of the former agent, even if it cannot offer evidence that its version is likely to be better. It may be that neither agent can persuade the other to adapt their ontology, and then communication will fail. If this happens, an agent that was initially unwilling to compromise, if it has a vested interest in producing a positive outcome from the communication, may become willing. This might happen in the situation encountered within the ORS: where the agent wishes a service to be provided by the other agent. In the current system, service-providing agents do not have any incentive to alter their ontologies to assist PAs. However, incentives could be introduced. Perhaps agents receive payment for performing services, and would therefore be willing to refine their ontologies to be able to perform them. Alternatively, it might be standard, or even enforced, behaviour in a service-providing agent that it should make some effort to comply with the demands of PAs by refining its ontology where necessary. This would be especially likely if payment were involved as not only, as mentioned above, would it benefit from performing the service, but also it would be more attractive as an agent for performing this service as opposed to other agents with the same capability, because it makes it easy for the PA to deal with it. As in human interaction, helpful service-providing agents would naturally be preferred over unhelpful agents.

In order to determine whether it was prepared to refine a particular part of its ontology, agents might have some kind of mark-up in their ontology concerning their willingness

to compromise over certain parts of the ontology. If it is a service-providing agent, it may be the authority on the circumstances in which an action can be performed. It would thus not be willing to change its opinion on this even to accommodate a fee-paying agent, because it has more faith in its opinion than that of a customer agent. If, for example, it refines its relevant action rule to accommodate a customer agent, it may result in a rule that does not describe the situation in which the action can be performed, and it will no longer be able to provide the service. Agents might also have particular facts on which they consider themselves to be the best authority, or more generally, predicates which they are unwilling to refine. This is related to the issue of epistemic entrenchment, discussed in Section 3.2.2.

So far, we have discussed the possibility of agents negotiating over which one of them is willing to be the one to refine its ontology. Another possibility is that they both refine their ontologies towards some middle point. This sort of negotiation would be rather more complicated. An agent who has detected a mismatch may begin the negotiations by announcing that it is unwilling to fully refine its ontology to cover this mismatch, but that it is prepared to refine it to a certain point, which would be stated. The other agent can then work out if it is willing to refine its ontology to this middle point and, if not, find some point it is willing to refine its ontology to and present this to the other agent. This process would terminate if they could find a common point that they are both prepared to meet, or if they have suggested and had rejected all the middle ground they are prepared to reach. For many refinements, this kind of negotiation would not be necessary. For example, if one agent has a binary predicate and the other a ternary, then either the first performs propositional anti-abstraction or the second performs propositional abstraction; there is no way they can find any middle ground between these two cases. However, for some refinements this might be possible: for example, if one agent had a binary predicate and the other had a 4-arity predicate, they might both agree to refine their ontologies to reach a ternary predicate. This method of negotiation would also be more relevant if the system were capable of dealing with

multiple refinements, as discussed in Section 8.4.4; this may lead to more complex situations in which there were many ways in which refinement could be done, and thus negotiation is more necessary and more difficult.

The major limitation of this approach is that it could only work when all of the agents involved were using ORS or a single system. Assuming that all agents one encounters have access to ORS severely limits its applicability. We want agents to be able to make use of ORS in a large and complex environment, where they cannot make many assumptions or force restrictions on the agents that they encounter. One approach to this would be to equip agents with the ability to negotiate over refinement with other agents, and to use this ability when they encounter agents that can also perform this, which would be agents that have access to this updated version of ORS. When they encounter agents that do not have this ability, they would then be able to refine their ontologies without negotiation.

8.4.3 Finding Anomalous Agents

A useful addition to ORS would be the ability to differentiate to some degree as to which agent it is desirable to interact with. Currently in ORS, if there is more than one agent that can perform a task, it chooses the agent merely on the basis of which was listed first in the meta-ontology. However, if many plans were executed in the same domain, thus requiring frequent interactions with the same agents, it would be worth differentiating between agents in a more intelligent fashion.

A simple approach would be to look back over past interactions with that agent and examine how many times the agent had successfully performed the action versus how many times it had failed, and, when it did perform it successfully, how many refinements were necessary to allow this. Statistics could be built up for each agent, and these could be compared to see which the PA should chose to interact with. If a service-

providing agent refused to perform actions under circumstances in which other agents would perform that service, this may also indicate that that agent should be avoided.

A more complex approach would be to observe the refinements that had been performed in response to an agent, and to examine whether any of these had had to be undone at a later point during communication with another agent. It might be that some agents are very similar to the PA and some are rather different. At least in the short term, it is in the PA's interests to interact with the agents that are more similar; however, if this restricted the PA to a small subset of agents, it might ultimately be worthwhile trying to attempt to interact with the majority of agents that are not so similar to the PA. If the PA refines an ontological object to match that of a first agent, and then refines it back to its original state to match that of a second agent, this implies that this part of the second agent's ontology is closer to the PA's ontology than first agent's is. If this happens repeatedly, it is clearly easier for the PA to deal with the second agent where possible, and avoid all unnecessary interaction with the first agent. Additionally, if agents charge more for their services than other agents providing similar services, this may be a cause to avoid them.

8.4.4 Dealing with Compound Mismatches

One obvious limitation of ORS is that it can only deal with refinements one at a time: if an ontological object differs in two ways, then one way is identified first and refined and then, when the updated object still leads to plan execution failure, the other mismatch is identified and refined. We assume that treating the mismatches as individuals will not affect our ability to diagnose and refine them, and we assume this because it restricts the complexity of the situation with which ORS has to deal. However, this is not necessarily a valid assumption.

A simple example of how a compound mismatch would confuse diagnosis is the case of

predicate that had had the class of an argument changed to a superclass, and had a new argument added; thus the correct refinement would be domain abstraction and propositional anti-abstraction. The diagnostic algorithm checks first to see if mismatched predicates with the same name have the same arity, and if not, diagnoses propositional refinement. However, in order to fully diagnose how this should be refined, the position of the extra argument needs to be identified, and this is done by matching the classes of the arguments of the two predicates and finding the position of the unmatched argument. The fact that the classes would not match would cause this process to fail. We can see in this situation that additions that could be made to the diagnostic algorithm might help. For example, if the classes do not match properly, the diagnostic algorithm could begin to look for subclass relations between them, and thus diagnose the compound mismatch appropriately.

In order to equip ORS to deal with compound mismatches, research would need to be done into exactly what sort of combinations of mismatches we might expect, and how these could be identified. Theoretically, there is no limit on the number of combinations we could encounter, because each type of mismatch could be applied many times. This number would only be restricted by the size of the ontology. It would thus be very difficult to identify all the potential mismatches, and may be extremely hard or, in some cases, impossible to diagnose which was being encountered.

If the history of refinement and attempting refinement was available, then it might be possible to revisit places where diagnosis had been impossible after further refinements had been performed. It might be that this problem was impossible to diagnose because it was due to some kind of complex mismatch, but that later refinement had caused some part of that complex mismatch to be automatically refined, leaving a similar mismatch that could now be diagnosed, thus reducing a problem of compound mismatch to one of simple mismatch.

It seems clear that it would not be possible to extend ORS so that it could infallibly

diagnose and refine every possible combination of mismatches. However, it seems equally clear that it would be possible to extend the functionality of ORS to be able to diagnose and refine some mismatch combinations, and that this functionality could be extremely useful. Research could be done to determine what the most common kinds of compound mismatches would be, and diagnostic algorithms for those could be developed where this was possible.

8.4.5 Complex Planning

Currently, ORS deals only with the linear plans produced by Metric-FF. However, there has been much work in the field of planning to produce planners that can produce more complex plans (see Section 2.6). It may be useful to take advantage of this ability within ORS.

Some forms of planning, such as conditional planning, partial-order planning or hierarchical planning, would allow for situations in which the failure of an action to be performed would not inevitably lead to the abandonment of the entire plan, and thus to refinement and replanning. Instead, some alternative plan could be followed or developed from the point at which failure occurred, without the need to stop the process and begin again, as happens currently in ORS. However, it is unclear quite how much of an advantage this would be. Is the aim of ORS to bring two ontologies closer together, or is it to facilitate the achievement of goals? If we believe the former, then ontological mismatch should always be investigated, diagnosed and refined, even if it may be possible to reach the goal without doing this. However, the purpose of the system may be regarded not as simply converging differing ontologies, but as facilitating agents to reach their goals by refining their ontologies where necessary, and from this point of view ontological mismatches can be ignored if there are alternative ways to reach the goal. This latter philosophy may be more appropriate for a Semantic Web scenario,

where an agent may encounter many hundreds of other agents, who may all have ontologies that differ in some regard. Thus aligning the ontologies in general becomes an unreasonable goal; the focus must be on refining ontologies only as a means to facilitate problematic communication. In the latter case, more complex forms of planning could be useful, as they may reduce the necessity of ontology refinement.

However, if introducing complex planning could reduce the need for ontology refinement, it would not greatly alter the way in which ontology refinement was done when it was necessary. If, for example, all the possible routes in a partial-order plan had been explored, and each option ended in failure, then ontology refinement would be performed just as is currently done in the system, and planning would begin again with the new ontology. This change would clearly affect the planning system, and may require some alteration in the plan execution ability of the PA, but it need not affect the diagnostic and refinement aspects of the system. There would be scope for some extra complexity. For example, some additional diagnostic information could be gleaned from knowledge of other potential branches of a plan that failed, and perhaps some ability to decide which branch to try to repair if all of them fails should be added in. However, this would not be necessary to allow more complex planning, it merely provides a means of making the best possible use of the information complex planning would provide.

8.4.6 E-institutions

Another approach to agent communication that we have investigated is the possibility of conducting it within the electronic institution framework [Sierra et al., 1998, Vasconcelos, 2002, Esteva et al., 2001]. E-institutions control agent interactions by providing a framework in which the interactions occur. Certain roles are defined by an institution, and an agent entering the institution must take on at least one of the roles. Some of these roles are available for any agent entering the system; in an auction scenario, this

might be a *buyer* or *seller* role. Other roles can only be fulfilled by institution agents; for example, *auctioneer*. A dialogical framework outlines acceptable speech acts, and group meetings are defined using scenes. Certain roles are permitted in each scene: for example, an auction scene might have exactly one *auctioneer* role (fulfilled by an institution agent) and one or more *buyer* roles (fulfilled by external agents). Agents that are not fulfilling these roles are not allowed to participate in the scene. The protocol of a scene is the possible dialogues agents may have. Scenes are connected by the performative structure, which prescribes how agents move from scene to scene. Agents are only allowed to leave or enter scenes at certain points. Obligations and effects of actions are kept track of by the normative rules of the system.

We chose not to use e-institutions in ORS because we wanted to avoid directing effort away from the central issues of diagnosis and refinement, and because e-institutions provided a tighter control on agent ontologies than we were interested in. If we were to extend the agent communication aspect of ORS, then e-institutions may be something we would consider using. Certainly, a more important first step for the agent communication system would be to make it FIPA-compliant; this would be a much less difficult transition and would have more obvious benefits. There are some reasons why we might not consider e-institutions appropriate for an extended version of ORS. Firstly, e-institutions place strict controls on the agents entering the institution; in particular, they must have the same ontology. We would therefore be interested in a more relaxed version of a standard e-institution, where this stringent approach to e-institutions would be loosened. However, we would not wish to loosen this constraint to the extent where agents with any kind of ontology would be able to enter: ORS can only be used when agents have similar ontologies. We would therefore need to investigate how constraints could be developed such that agents would only be able to enter the institution if their ontologies were appropriately similar to the ontology used by the e-institution.

Adding the functionality of ORS might be useful to e-institutions, so that they would not have to place such stringent restrictions on agents entering the institution but instead allow some amount of leeway, so that agents with updated, out of date or altered versions of the correct ontology could make use of the e-institution, thus increasing its applicability. However, from the point of view of ORS, such work is not as obviously useful as some of the other issues discussed in this chapter, which would make ORS better able to perform in its most natural environment: the Semantic Web or similar large open-agent communication architectures.

8.4.7 Patching the Plan

Currently, ORS makes no attempt to update plans after plan execution failure is encountered and diagnosis and refinement performed. Instead, the existing plan is discarded and a completely new plan is formed from the updated ontology. As can be seen in 7.3, plans formed from updated ontologies are often very similar or even identical to the original plan. It might therefore be better to patch the plans so that the changes to the ontology were reflected in it, rather than forming a new plan.

One framework for creating this functionality could be the work on management of change carried out by Hutter and Autexier [Autexier et al., 2002, Autexier and Hutter, 2002,Hutter, 2000]. This work is focused on patching formal verifications of specifications after small changes have been made to them. It involves creating a *development graph* to track the dependencies within the specification, and this can be used to deduce which parts of the specifications will, or might, have been affected by the changes. Similar techniques could be developed to track the dependencies between a plan and its underlying ontology. In fact, this information is already provided to some extent by the plan justification; this could be altered so as to provide the relevant information. Applying these ideas to a planning context would be somewhat complicated by the non-monotonicity of planning: finding evidence that a precondition has been true

before an action is performed is not sufficient justification to prove that it is true when the action is performed: it must be shown that it has not been made false at any intermediate point. The resulting system would be similar to an assumption-based truth maintenance system (ATMS; see Section 3.2.4).

Forming plans that are only a few steps long, such as those illustrated in Section 7.3, is a fairly speedy process, and thus the benefits of patching plans may not outweigh the costs. However, if ORS were forming longer plans, which it is equipped to do, the balance may be tipped in the other direction. An additional benefit of this approach would be that it would not be necessary to produce new PDDL files from the updated ontology, and thus much effort in translation would also be spared; this is especially relevant if the ontology is complex, as translating to PDDL is currently the slowest part of running ORS.

A minimal implementation of this idea could be to check whether a plan produced from this updated ontology would be identical to the original plan; thereby avoiding the need either for replanning or for patching the plan. However, this is slightly more difficult than a first glance might suggest. Obvious changes to plans involve changing the action steps, but more subtle changes involve changing the arguments those action steps take. If, for example, propositional anti-abstraction is performed, this may not affect the plan steps of a plan but it will sometimes affect the arguments of one or more of the plan steps. This new argument will appear in the arguments of any action to which it is relevant. It may be that this argument already appears as part of that action, in which case there will be no change to the action step. On the other hand, it may not be already listed: in this case, there will be a change to the arguments of the actions. This kind of information would have to be calculated before it could be determined whether the plan needed to be altered or not. Some research would need to be done to determine whether such functionality would be of benefit to ORS.

8.5 Summary

The ideas discussed in this chapter could not be added to ORS without a considerable amount of work. We feel that the most important aspects to address are those that would prepare ORS for use on the Semantic Web, or within a large multi-agent system where the agents were diverse and owned by many different users. Ideas that would improve the performance of ORS but are not essential for its use in such a situation would be given a lower priority.

Chapter 9

Conclusions

Communication is an issue of increasing importance in Artificial Intelligence. A greater emphasis is being placed on the development of large-scale systems where huge numbers of users with different aims, locations and backgrounds can interact and communicate easily in areas where their interests overlap: the Semantic Web and the Grid are the two most well known examples of this. A prerequisite to the success of such systems is that they have to be easy to use; if it is only possible to use them by investing a large amount of time and effort into complying with their restrictions, potential users will be disinclined to become involved with them.

A crucial aspect of the ability to communicate, as discussed in Chapter 1, is the existence of shared language to describe the world. However, insisting on this clashes with the need to create open, usable systems where tight restrictions are not enforced. The problem of how to create this shared language from two representations that may not initially be identical, and to do this during runtime as the interactions are taking place, is thus an important function of any large-scale, multi-user system. We claim that an effective method of doing this is through dynamic ontology refinement.

In this thesis, we have outlined why we believe this to be a good approach, and explained how our specific approach differs from that taken by other people. We have

introduced our system ORS, and explained how the theory behind the implementation addresses this tension between the need to refrain from enforcing strict constraints and the need for mutual comprehension between agents. We have evaluated ORS in order to demonstrate that it is both usable and useful.

9.1 Contributions

We summarise the contributions of the thesis, both those that are achieved by fulfilling the hypothesis and aims of the project, outlined in Chapter 1, and those that have been achieved in addition to this, through the development of the subsystems and the functionality required to achieve the central aims.

9.1.1 Central Contribution

The hypothesis of the thesis was stated as:

> Using dynamic ontology refinement to locate and correct ontological mismatches between agents can enable successful communication which would otherwise be impossible.

We claim that this hypothesis is proved through the fulfilment of three aims, listed below. We revisit the aims of the project, and explain how these have been achieved.

- *To provide a framework in which agents with first-order, largely similar ontologies can diagnose ontological mismatches between them:*

  Chapter 5 describes the diagnostic process that allows agents to identify mismatches between their first-order ontologies. Figure 5.9 describes all of the potential mismatches between first-order theories, and highlights which of these ORS can always, or sometimes, or never deal with. The diagnostic algorithms

are designed for agents that are using plan failure and plan justification to deduce what mismatch they are encountering, and this method of detecting failure is not suitable for discovering all possible mismatches. Additionally, adding information into an ontology can be hard, as it is not always clear what this new information should be. We explain how these lead to the areas of the chart where we encounter mismatches that are sometimes or never diagnosable in ORS.

- *To integrate this framework into a system, ORS, that enables an environment where planning agents can use this ability to reach goals that would otherwise have been unreachable. This system must be fully automated:*

  Chapter 4 describes how the diagnostic capabilities of ORS are embedded in a fully automated system which facilitates this behaviour. It explains the contribution of each subsystem to the whole, and the manner in which they are integrated. Chapter 6 explains in more detail the theory and implementations behind these subsystems.

- *To evaluate these abilities against genuine examples of ontological mismatches, to demonstrate that these abilities are useful:*

  In Chapter 5 we outlined the space of possible mismatches in first-order ontologies, explaining which ORS could and could not identify. Chapter 7 evaluates the performance of ORS with reference to genuine ontological mismatches from different versions of ontologies. The results highlight some areas where ORS could be improved and extended, and also illustrate the limitations of performing this refinement process within a planning context. Nevertheless, the results were mostly positive for ORS, indicating that this kind of matching is not only theoretically useful, but can also deal with real mismatches. Unfortunately, the development of the Semantic Web and similar systems is not complete enough to give us access to many different kinds of ontologies and ontological mismatches; thus the evaluation of ORS is not as thorough as we would wish. We hope to ex-

tend this evaluation as more examples of such ontologies become available and as ORS is developed to become increasingly sophisticated; this evaluation and development is envisaged as an interleaved process.

The realisation of these aims has allowed us to produce a prototype of a system that allows dynamic, fully automated ontology matching during runtime. Chapter 3 and Section 7.5 explain how this need is not already being met in a satisfactory way by the current technology, and motivate why this is an important ability. We feel, therefore, that we have contributed to the field by exploring ways in which this might be done, and producing a working system that demonstrates that these ideas can work in practise on real ontologies. The system is not currently suitable for use on the Semantic Web; we have explained in Chapter 4 why this was not a reasonable aim for the project, and detail in Chapter 8 the extra work that would be required to facilitate this. We believe that more investment in ontology matching of this sort, which, as we explain in Section 7.5, is quite different from standard ontology mapping, would bring the existence of a fully-realised Semantic Web a step closer.

9.1.2 Additional Contributions

The aim and focus of this project has been the investigation of solutions to ontology mismatch, with diagnosis and refinement as the key components of the resulting system. However, we believe this project has made additional contributions that are not directly related to this issue, but instead were developed as part of the sub-systems of ORS.

The main area in which this lies is in the interaction between ontologies and planning. Since we have placed our ontology refinement system in a planning domain, we have had to address this problem. This is a field that has traditionally been ignored by the planning community, but which is beginning to generate more interest. As the

importance of planning for agents in large, multi-agent systems such as the Semantic Web becomes clearer, this issue is being addressed more urgently by both the planning and the ontology communities.

We have contributed to this field in two particular ways:

- **The Plan Deconstructor**

  As we have discussed in Chapter 4 and Section 6.3, one of the drawbacks of the efficient search techniques employed by modern planners is that there is very little feedback about how the plan produced is related to the underlying ontology. This is acceptable if successful plan execution is inevitable, but it means that there is no scope for learning about the domain through plan failure in a domain in which plan execution is not assured, due to the unpredictability of the domain or the potential errors inherent in representing a domain. We have solved this problem in ORS through the use of the plan deconstructor, which we believe to be a unique approach to the problem. This work was presented to the planning community in [McNeill et al., 2003b].

- **Translation from KIF to PDDL**

  Another significant issue in this field is that representations required for modern planners are significantly different from standard ontological representations. This is because it is important in planning to limit the expressivity of the representation, as otherwise the search problems are enormous. Standard ontological representations, on the other hand, are usually quite expressive, as this allows more complex information about the domain to be encoded. We have dealt with this problem by creating a translation process between restricted KIF and PDDL. Since we are translating from a more expressive representation to a less expressive one, there is the potential for some information to be lost. We believe that this loss of expressivity is acceptable because the expressivity needs of the two domains are too different to be combined. We cope with this loss of expressivity

in ORS by using the PDDL ontology only for planning, and retaining the KIF ontology. The work was presented to the planning community in [McNeill et al., 2004a] and [McNeill et al., 2005].

9.1.3 Future Contributions

ORS currently demonstrates the applicability of the theory to the problem of ontological mismatch in real-world domains. However, it is not yet sufficiently adaptable and sophisticated to be used on a system such as the Semantic Web. We have outlined in Chapter 8 how we would like to extend ORS so that it would be possible for it to operate in such a domain, and to operate more successfully than it is currently able to. We envisage dynamic ontology refinement systems such as ORS having an important role to play in the Semantic Web and other large multi-user systems.

Appendix A

Example Ontology

Excerpts from Semantic Web Conference Ontology [1]

```
(In-Package "ONTOLINGUA-USER")


(Define-Ontology
     Sem-Web-Conf
     (Frame-Ontology)
   "Not supplied yet."
   :Io-Package
   "ONTOLINGUA-USER")


(In-Ontology (Quote Sem-Web-Conf))
```

CLASS HIERARCHY

```
;;; Action
```

[1] This is the ontology with which much of the development was done. The full ontology, the meta-ontology, and all the ontologies used for evaluation (those mentioned in Chapter 7) can be found on the project webpage (`http://dream.inf.ed.ac.uk/projects/dor/`)

```
(Define-Class Action (?X) :Def (And (Thing ?X)))

;;; Agent
(Define-Class Agent (?X) :Def (And (Thing ?X)))

;;; Confirmation-Number
(Define-Class Confirmation-Number (?X) :Def (And (Thing ?X)))

;;; Event
(Define-Class Event (?X) :Def (And (Thing ?X)))

  ;;; Conference
  (Define-Class Conference (?X) :Def (And (Event ?X)))

;;; Object
(Define-Class Object (?X) :Def (And (Thing ?X)))

  ;;; Paper
 (Define-Class Paper (?X) :Def (And (Object ?X)))

    ;;; Dvi-Paper
   (Define-Class Dvi-Paper (?X) :Def (And (Paper ?X)))

    ;;; Pdf-Paper
   (Define-Class Pdf-Paper (?X) :Def (And (Paper ?X)))

    ;;; Ps-Paper
   (Define-Class Ps-Paper (?X) :Def (And (Paper ?X)))
```

```
;;; Place
(Define-Class Place (?X) :Def (And (Thing ?X)))

  ;;; City
  (Define-Class City (?X) :Def (And (Place ?X)))
```

RELATIONS

```
;;; Has-Paper
(Define-Relation Has-Paper (?Agent ?Paper ?Situation)
        :Def (And (Agent ?Agent) (Paper ?Paper) (Sit-Var ?Situation)))


;;; Location
(Define-Relation Location (?Thing ?Place ?Situation)
        :Def (And (Thing ?Thing) (Place ?Place) (Sit-Var ?Situation)))
```

FUNCTIONS

```
;;; Accepted-Paper
(Define-Function Accepted-Paper
          (?Agent-0 ?Paper-1 ?Conference-2 ?Situation) :-> ?Value
          :Def (And (Agent ?Agent-0)
                    (Paper ?Paper-1)
                    (Conference ?Conference-2)
                    (Confirmation-Number ?Value)
                    (Sit-Var ?Situation)))
```

```
;;; Flight

(Define-Function Flight (?Place-0 ?Place-1 ?Situation) :-> ?Value
            :Def (And (Place ?Place-0)
                      (Place ?Place-1) (Number ?Value)
                      (Sit-Var ?Situation)))

;;; Has-Accom

(Define-Function Has-Accom
          (?Agent-0 ?Conference-1 ?Situation) :-> ?Value
          :Def (And (Agent ?Agent-0)
                    (Conference ?Conference-1)
                    (Confirmation-Number ?Value)
                    (Sit-Var ?Situation)))

;;; Has-Ticket

(Define-Function Has-Ticket (?Agent-0 ?Situation) :-> ?Value
          :Def (And (Agent ?Agent-0) (Confirmation-Number ?Value)
                    (Sit-Var ?Situation)))

;;; Money

(Define-Function Money (?Agent-0 ?Situation) :-> ?Value
          :Def (And (Agent ?Agent-0) (Number ?Value)
                    (Sit-Var ?Situation)))
```

```
;;; Registered

(Define-Function Registered
          (?Agent-0 ?Conference-1 ?Situation) :-> ?Value
          :Def (And (Agent ?Agent-0) (Conference ?Conference-1)
                    (Confirmation-Number ?Value)
                    (Sit-Var ?Situation)))
```

INDIVIDUALS

```
;;; Cade
(Define-Frame Cade
        :Own-Slots ((Documentation) (Instance-Of Conference)
                    (Location Miami [(Start)])
                    (Registration-Fee 200 [(Start)]))
        :Axioms ((Accomodation-Info Cade 50 [(Start)])))

;;; Edinburgh
(Define-Frame Edinburgh
          :Own-Slots ((Documentation) (Instance-Of City))
          :Axioms ((Flight Edinburgh Miami 300 [(Start)])))

;;; Isabelle-Paper-Dvi
(Define-Individual Isabelle-Paper-Dvi (Dvi-Paper) )

;;; Isabelle-Paper-Ps
(Define-Individual Isabelle-Paper-Ps (Ps-Paper) )
```

```
;;; Lucas
(Define-Frame Lucas
        :Own-Slots ((Documentation ) (Has-Paper Isabelle-Paper-Dvi)
                    (Instance-Of Agent) (Location Edinburgh [(Start)]))
        :Axioms ((Money Lucas 1000 [(Start)])))

;;; Miami
(Define-Individual Miami (City) )

;;; Pseudo-Var
(Define-Individual Pseudo-Var (Confirmation-Number) )
```

AXIOMS

```
;;; Book-Accom
(Define-Axiom Book-Accom  :=
  (=>
   (And (Accomodation-Info ?Conference ?Cost ?Sit1)
        (Money ?Agent ?Amount ?Sit1) (< ?Cost ?Amount))
   (And (Has-Accom ?Agent Pseudo-Var ?Sit2)
        (= ?Newamount (- ?Amount ?Cost))
        (Money ?Agent ?Newamount ?Sit2)
         (Not (Money ?Agent ?Amount ?Sit2)))))

;;; Book-Flight
(Define-Axiom Book-Flight  :=
  (=>
   (And (Location ?Agent ?Agent-Loc ?Sit1)
        (Location ?Conference ?Conf-Loc ?Sit1)
```

```
        (Flight ?Agent-Loc ?Conf-Loc ?Price ?Sit1)
        (Money ?Agent ?Amount ?Sit1)
        (< ?Price ?Amount))
   (And (Has-Ticket ?Agent Pseudo-Var ?Sit2)
        (Money ?Agent ?Newamount ?Sit2)
        (= ?Newamount (- ?Amount ?Price))
        (Not (Money ?Agent ?Amount ?Sit2)))))

;;; Convert-Paper
(Define-Axiom Convert-Paper  :=
  (=>
   (And (Has-Paper ?Agent ?Paper-Dvi ?Sit1)
        (Dvi-Paper ?Paper-Dvi ?Sit1))
   (And (Has-Paper ?Agent ?Paper-Ps ?Sit2)
        (Ps-Paper ?Paper-Ps ?Sit2))))

;;; Register
(Define-Axiom Register  :=
  (=>
   (And (Accepted-Paper ?Agent ?Paper ?Conference Pseudo-Var ?Sit1)
        (Money ?Agent ?Amount ?Sit1)
        (Registration-Fee ?Conference ?Cost ?Sit1)
        (< ?Cost ?Amount))
   (And (Registered ?Agent ?Conference Pseudo-Var ?Sit2)
        (Money ?Agent ?Newamount ?Sit2)
        (= ?Newamount (-?Amount ?Cost))
        (Not (Money ?Agent ?Amount ?Sit2)))))
```

Appendix B

Refinement Code

```
refine(Type,Info) :-
        checkTypes,
        assert(newTypes([])),
        see('ont.in'),tell('ont.out'),
        performRef(Type,Info),
        newTypes(TypesList),
        addClasses(TypesList),
        told,seen,
        copyFiles.
```

refine/2 will read in the old ontology and write to a new ontology, `ont.out`. A list of classes (types) in the old ontology is built, so that it is possible to tell if classes which appear as part of a refinement are already in the class hierarchy, or if they need to be added. A class list, which includes information about new classes together with their superclasses, is built up and these are added to the new ontology after the process has been completed. *performRef/2* is the command that actually performs the refinement. After the whole process is completed, the new ontology is copied onto the old ontology, so that the name of the ontology remains constant.

```
performRef(Type,Info) :-
        readLineRef(Line),
        (   extractLast(Line,end,Rest),nl,
            checkLine(Type,Info,Rest)
        ;
            checkLine(Type,Info,Line),
            performRef(Type,Info)
        ).
```

performRef/2 reads the ontology line by line (since *refine/2* sent the input stream to `ont.in`, a read instruction will find the first line of that which has not already been read). The end of file indicator is checked for, and if this is found, the rest of the line is processed and the procedure terminates. If not, the line is processed and then the rest of the ontology is read. *checkLine/3* will analyse a particular line to see if it is a line that needs to be changed as part of the refinement process. If so, this is carried out (*processLine/3*). If not, it is copied verbatim to the ontology (*copyLine/1*).

An example of the code for precondition anti-abstraction is given below. This is the easiest refinement to perform, and thus the code is the most concise. Note that in precondition anti-abstraction, we only need be concerned with one line of the ontology: the line defining the relevant action. Some of the other refinements are more complex and require us to look at several lines of the ontology. For example, propositional refinement requires us to alter not only the definition of the relation, but also every occurrence of it.

```
processLine(precond,[Rule,Precond],Line):-
```

first, check if the line is a rule

```
        name('(Define-Axiom ', DefAx),
```

```
matchExpression([],DefAx,AxNameAndDef,Line),
```

if so, find the ASCII name of the rule, append a space and see if this matches the name of this particular rule

```
name(Rule,RuleChars),
name(' ',Space),
append(RuleChars,Space,WholeRuleChars),
matchExpression([],WholeRuleChars,Def,AxNameAndDef),
```

if it does, we must find where the preconditions start

```
name('=> (And ',PostImp),
(   matchExpression(Comment,PostImp,PrecondsAndRest,Def)
;
    name('=> ',SinglePostImp),
    matchExpression(Comment,SinglePostImp,PrecondsAndRest,Def)
),
append(Space,PrecondsAndRest,PrecondsAndSpace),
```

then we convert the precond into ASCII and append it

```
name(Precond,PrecondChars),
append(PrecondChars,PrecondsAndSpace,NewPrecondsAndRest),
```

then we rebuild what we have taken apart

```
append(PostImp,NewPrecondsAndRest,NewDef),
append(Comment,NewDef,NewDef2),
append(WholeRuleChars,NewDef2,NewAxNameAndDef),
append(DefAx,NewAxNameAndDef,NewLineChars),
```

we then convert it into text and write it to the ontology file

```
        name(NewLine,NewLineChars),
        write(NewLine).
```

Appendix C

Output of System

The output shown here is produced by two of the six ontologies we have run on the system, edited for readability and conciseness. For the complete output of all the ontologies, see the project website[1].

C.1 AKT Ontology

```
| ?- plan.
consulting the translator ...
consulting the plan finder ...
consulting the plan deconstructor ...
consulting the ontology updater ...
consulting the diagnostic algorithm ...
consulting the refinement system ...

What is the goal?
```

[1] http://dream.inf.ed.ac.uk/projects/dor/

```
|: memberOrganization(researchAgent,refinementSig)

translating ...
translation done
need to find a plan ...

This is the plan:
[becomeMemberOrganization(refinementSig,researchAgent,
 schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)]

deconstructing the plan ...
executing the plan ...

im going to ask membershipAgent to perform
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)
for me

plan has failed at
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)

The KIF ontology has been updated

Im diagnosing what the problem is ...

I received a query about hasEmailAddress(researchAgent,_297427), which
```

```
I was not expecting to be asked about.
No preconds have the same name as hasEmailAddress.

diagnosis: precondition anti-abstraction
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)
requires an extra precondition: hasEmailAddress(researchAgent,_297427)
WARNING: this is a guess

I know that researchAgent is of class agent
I know that pseudoVar is of class confirmationNumber
this is the precond I need to add:
(Has-Email-Address ?Agent ?Confirmation-Number)

the appropriate refinement has been performed


goal is: memberOrganization(researchAgent,refinementSig)

translating ...
translation done
need to find a plan ...

This is the plan:
[generateEmailAddress(researchAgent,schoolOfInformaticsAtEdinburgh),
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)]
```

```
deconstructing the plan ...
executing the plan ...

I'm going to ask emailAgent to perform
generateEmailAddress(researchAgent,
schoolOfInformaticsAtEdinburgh)
for me

generateEmailAddress(researchAgent,schoolOfInformaticsAtEdinburgh)
completed satisfactorily

I'm going to ask membershipAgent to perform
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)
for me

plan has failed at
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,subscriptionFee,ontologyRefinement)

The KIF ontology has been updated

I'm diagnosing what the problem is ...

I received a query about memberAcademicUnit(researchAgent,
schoolOfInformaticsAtEdinburgh,ukCsDepts),
which I was not expecting to be asked about.
memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh,ukCsDepts)
```

```
has the same name as the precondition
memberAcademicUnit(researchAgent,schoolOfInformaticsAtEdinburgh)
They have different arity (3 and 2)

diagnosis: propositional anti-abstraction
my arguments are of these classes : [academicUnit,agent]
membershipAgent tells me that ukCsDepts is of class
educationalOrganizationalUnit

membershipAgent tells me that educationalOrganizationalUnit is of
class organizationUnit

the appropriate refinement has been performed


goal is: memberOrganization(researchAgent,refinementSig)

translating ...
translation done
need to find a plan ...

This is the plan:
[becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,schoolOfInformaticsAtEdinburgh,
subscriptionFee,ontologyRefinement)]

deconstructing the plan ...
executing the plan ...
```

```
I'm going to ask membershipAgent to perform
becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,schoolOfInformaticsAtEdinburgh,
subscriptionFee,ontologyRefinement)
for me

becomeMemberOrganization(refinementSig,researchAgent,
schoolOfInformaticsAtEdinburgh,schoolOfInformaticsAtEdinburgh,
subscriptionFee,ontologyRefinement) completed satisfactorily

The KIF ontology has been updated

The plan is completed
```

C.2 Lift Scheduling Ontology

```
| ?- plan.
consulting the translator ...
consulting the plan finder ...
consulting the plan deconstructor ...
consulting the ontology updater ...
consulting the diagnostic algorithm ...
consulting the refinement system ...

| ?- and(served(personOne),served(personTwo),served(personThree))
```

```
translating ...
translation done
need to find a plan ...

This is the plan:
[serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer),serveLiftCustomer(personTwo,floorZero,
liftOne,securityCleared,floorOne,onsiteEngineer),
serveLiftCustomer(personThree,floorTwo,liftOne,securityCleared,
floorZero,onsiteEngineer)]

deconstructing the plan ...
executing the plan ...

I'm going to ask serveAgent to perform
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer)
for me

plan has failed at
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer)

The KIF ontology has been updated

I'm diagnosing what the problem is ...

this plan failed immediately after a request was made to perform
```

```
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer)
serveAgent says that he can perform this task
diagnosis: problem precond: inService(liftOne,onsiteEngineer)
inService(liftOne,onsiteEngineer) is an original fact in my ontology

the appropriate refinement has been performed


goal is: and(served(personOne),served(personTwo),served(personThree))

translating ...
translation done
need to find a plan ...

This is the plan:
[moveEmptyLift(floorOne,liftOne,floorZero),serviceLift(liftOne),
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer),serveLiftCustomer(personTwo,floorZero,
liftOne,securityCleared,floorOne,onsiteEngineer),
serveLiftCustomer(personThree,floorTwo,liftOne,securityCleared,
floorZero,onsiteEngineer)]

deconstructing the plan ...
executing the plan ...

I'm going to ask moveAgent to perform
moveEmptyLift(floorOne,liftOne,floorZero)
```

```
for me

moveEmptyLift(floorOne,liftOne,floorZero) completed satisfactorily

I'm going to ask serviceAgent to perform
serviceLift(liftOne)
for me

serviceLift(liftOne) completed satisfactorily

I'm going to ask serveAgent to perform
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer)
for me

plan has failed at serveLiftCustomer(personOne,floorOne,liftOne,
securityCleared,floorZero,onsiteEngineer)

The KIF ontology has been updated

I'm diagnosing what the problem is ...

There seems to be a problem with hasAccess(personOne,floorOne)
although I knew that I would be asked about it.

this is fully instantiated: therefore, I must negate my expectations
about hasAccess(personOne,floorOne)
```

```
have to unnegate this

the appropriate refinement has been performed


goal is: and(served(personOne),served(personTwo),served(personThree))

translating ...
translation done
need to find a plan ...

This is the plan:
[grantAccess(floorOne,personOne),grantAccess(floorZero,personTwo),
grantAccess(floorTwo,personThree),serveLiftCustomer(personOne,floorOne,
liftOne,securityCleared,floorZero,onsiteEngineer),
serveLiftCustomer(personTwo,floorZero,liftOne,securityCleared,
floorOne,onsiteEngineer),serveLiftCustomer(personThree,floorTwo,
liftOne,securityCleared,floorZero,onsiteEngineer)]

deconstructing the plan ...
executing the plan ...

I'm going to ask accessAgent to perform
grantAccess(floorOne,personOne) for me

grantAccess(floorOne,personOne) completed satisfactorily

I'm going to ask accessAgent to perform
```

```
grantAccess(floorZero,personTwo) for me

grantAccess(floorZero,personTwo) completed satisfactorily

I'm going to ask accessAgent to perform
grantAccess(floorTwo,personThree) for me

grantAccess(floorTwo,personThree) completed satisfactorily

I'm going to ask serveAgent to perform
serveLiftCustomer(personOne,floorOne,liftOne,securityCleared,
floorZero,onsiteEngineer) for me

plan has failed at serveLiftCustomer(personOne,floorOne,liftOne,
securityCleared,floorZero,onsiteEngineer)

The KIF ontology has been updated

I'm diagnosing what the problem is ...

I received a query about call(personOne,liftOne),
which I was not expecting to be asked about.
call(personOne,liftOne) has the same name as the
precondition call(personOne,liftOne,securityCleared)
They have different arity (2 and 3)

diagnosis: propositional abstraction.
my arguments are of these classes : [authorization,lift,person]
```

```
the appropriate refinement has been performed


goal is: and(served(personOne),served(personTwo),served(personThree))

translating ...
translation done
need to find a plan ...

This is the plan:
[serveLiftCustomer(personOne,floorOne,liftOne,floorZero,onsiteEngineer),
serveLiftCustomer(personTwo,floorZero,liftOne,floorOne,onsiteEngineer),
serveLiftCustomer(personThree,floorTwo,liftOne,floorZero,onsiteEngineer)]

deconstructing the plan ...
executing the plan ...

I'm going to ask serveAgent to perform
serveLiftCustomer(personOne,floorOne,liftOne,floorZero,onsiteEngineer)
for me

serveLiftCustomer(personOne,floorOne,liftOne,floorZero,onsiteEngineer)
completed satisfactorily

I'm going to ask serveAgent to perform serveLiftCustomer(personTwo,
floorZero,liftOne,floorOne,onsiteEngineer) for me
```

```
serveLiftCustomer(personTwo,floorZero,liftOne,floorOne,onsiteEngineer)
completed satisfactorily

I'm going to ask serveAgent to perform serveLiftCustomer(personThree,
floorTwo,liftOne,floorZero,onsiteEngineer) for me
serveLiftCustomer(personThree,floorTwo,liftOne,floorZero,onsiteEngineer)

completed satisfactorily

The KIF ontology has been updated

The plan is completed
```

Appendix D

Glossary

Abstraction - removal of detail from the signature and, consequently, the theory.

Anti-abstraction - addition of detail to the signature and, consequently, the theory.

Domain refinement - alteration of the class of an argument of a predicate. This includes domain abstraction and domain anti-abstraction.

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

Meta-Ontology - higher-level ontology that provides information about objects in the standard ontology.

Metric-FF - planner used by ORS.

Ontology - domain information that consists of a signature or language in which the domain can be described and a theory or set of axioms written in the signature terms.

Ontology mismatch - terms in two different ontologies which refer to the same thing are not identically expressed, or an ontological object expressed in one is not expressed in the other.

Ontology refinement - alteration of a single term or multiple terms in an ontology.

ORS - Ontology Refinement System: the system described in this project.

PA - Planning Agent: an agent that is engaged in plan execution, considered in this project to be the agent that is performing, when necessary, dynamic ontology refinement.

PDDL - propositional language used in planning.

Plan deconstruction - meta-interpretation of a plan with reference to the underlying ontology.

Plan justification - the link between the plan and the underlying ontology that is produced by plan deconstruction.

Precondition refinement - alteration of the number of preconditions that a rule has. This includes precondition abstraction and anti-abstraction.

Predicate refinement - alteration of the name of a predicate. This includes predicate abstraction and anti-abstraction.

Propositional refinement - alteration of the number of arguments a predicate has. This includes propositional abstraction and anti-abstraction.

Pseudo-variables - used in planning in place of ordinary variables, which are not permitted in PDDL. A pseudo-variable is interpreted by the planner as an individual, but by the PA as a variable.

Service-providing Agent - an agent that is able to perform services for other agents under specific circumstances. The PA performs plan steps through interaction with service-providing agents.

Signature - description of the representation language in which the ontology is written: what predicates exist, what arity they have, what classes their arguments have, the class hierarchy and so on.

Theory - when used in the context of being part of an ontology, it is a reference to the formulae written in the representation language (or signature).

Bibliography

[AKT, 2002] AKT (2002). Akt project. http://www.aktors.org.

[Amarel, 1968] Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D., editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press.

[Antoniou et al., 2005] Antoniou, G., Franconi, E., and van Harmelen, F. (2005). Introduction to semantic web ontology languages. In Eisinger, N. and Małuszyński, J., editors, *Reasoning Web, Proceedings of the Summer School, Malta*, number 3564 in Lecture Notes in Computer Science, Berlin, Heidelberg, New York, Tokyo. Springer-Verlag.

[Antoniou and van Harmelen, 2004] Antoniou, G. and van Harmelen, F. (2004). *A Semantic Web Primer*. The MIT Press.

[Austin, 1962] Austin, J. L. (1962). *How to do Things with Words*. Claredon, Oxford, UK.

[Autexier and Hutter, 2002] Autexier, S. and Hutter, D. (2002). Maintenance of formal software developments by stratified verification. In Baaz, M. and Voronkov, A., editors, *Proceedings of the 9th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2002, Springer LNCS 2514, Tbilisi, Georgia.*

[Autexier et al., 2002] Autexier, S., Hutter, D., Mossakowski, T., and Schairer, A. (2002). The development graph manager MAYA. In Kirchner, H., editor, *Proceedings of 9th International Conference on Algebraic Methodology And Software Technology (AMAST'02)*. Springer Verlag.

[Banerjee et al., 1987] Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. (1987). Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 311–322, New York, NY, USA. ACM Press.

[Belief Revision, 2005] Belief Revision (2005). http://www.beliefrevision.org/.

[Berners-Lee et al., 2001] Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, 284(5):34–43.

[Besana et al., 2005] Besana, P., Robertson, D., and Rovatsos, M. (2005). Exploiting interaction contexts in peer-to-peer ontology mapping. In *Proceedings of the Second International Workshop of Peer-to-Peer Knowledge Management (P2PKM)*, San Diego, CA, USA.

[Blum and Furst, 1995] Blum, A. and Furst, M. (1995). Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642.

[Borst et al., 1997] Borst, P., Akkermans, H., and Top, J. (1997). Engineering ontologies. *Internation Journal of Human-Computer Studies*, 46(2-3):365–406.

[Bouquet et al., 2004] Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini1, L., and Stuckenschmidt, H. (2004). Contextualizing ontologies. *Journal of Web Semantics*, 1(4):24.

[Bundy, 1991] Bundy, A. (1991). A science of reasoning. In Lassez, J.-L. and Plotkin, G., editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press.

[Bundy, 1998] Bundy, A. (1998). The use of explicit plans to guide proofs. In Lusk, E. and Overbeek, R., editors, *Proceedings of CADE-9*, pages 111–120. Springer-Verlag Lecture Notes in Computer Science No. 310.

[Bundy et al., 1991] Bundy, A., Grosse, G., and Brna, P. (1991). A recursive techniques editor for prolog. In *Instructional Science*, volume 20(2/3), pages 135–172.

[Campbell and Shapiro, 1998] Campbell, A. E. and Shapiro, S. C. (1998). Algorithms for ontological mediation. In *COLING-ACL Workshop*, pages 102–107.

[Chandrasekaran et al., 1999] Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R. (1999). What are ontologies, and why do we need them? *IEEE Intelligent Systems*, 14(1):20–26.

[Clark and Uschold, 2002] Clark, P. and Uschold, M. (2002). The many faces of the Semantic Web. *IEEE Intelligent Systems*, 17(2):70–79.

[Colton, 2002] Colton, S. (2002). HR. http://www.doc.ic.ac.uk/ sgc/hr/.

[Console and Dressler, 1999] Console, L. and Dressler, O. (1999). Model-based diagnosis in the real world: Lessons learned and challenges remaining. In *Proceedings of the sixteenth International Joint Conference in Artificial Intelligence (IJCAI'99)*, pages 1393–1400, Stockholm, Sweden.

[Cuenca Grau et al., 2004] Cuenca Grau, B., Parsia, B., and Sirin, E. (2004). Working with multiple ontologies on the semantic web. In *Proceedings of the Third Internatonal Semantic Web Conference (ISWC2004). Volume 3298 Lecture Notes in Computer Science.*

[da Silva et al., 2002] da Silva, F. S. C., Vasconcelos, W. W., Robertson, D. S., Brilhante, V., de Melo, A. C., Finger, M., and Agust, J. (2002). On the insufficiency of ontologies: Problems in knowledge sharing and alternative solutions. *Knowledge Based Systems*, 15(3):147–167.

[DAML, 2002] DAML (2002). Daml.org. http://www.daml.org.

[DAML-S, 2002] DAML-S (2002). Daml services. http://www.daml.org/services/.

[DAML+OIL, 2002] DAML+OIL (2002). http://www.daml.org/2001/03/daml+oil-index.html/.

[de Kleer and Williams, 1987] de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130.

[DL, 2004] DL (2004). Description logics. http://www.ida.liu.se/labs/iislab/people/patla/DL/.

[Doan et al., 2003] Doan, A., Madhavan, J., Dhamankar, R., Domingos, P., and Halevy, A. (2003). Learning to match ontologies on the semantic web. *The VLDB Journal*, 12(4):303–319.

[Durfee, 1999] Durfee, E. H. (1999). Distributed problem solving and planning. In Weiß, G., editor, *A Modern Approach to Distributed Artificial Intelligence*, chapter 3. The MIT Press, San Francisco, CA.

[Ehrig and Sure, 2004] Ehrig, M. and Sure, Y. (2004). Ontology mapping - an integrated approach. In *ESWS*, pages 76–91.

[Ehrig et al., 2003] Ehrig, M., Tempich, C., Broekstra, J., van Harmelen, F., Sabou, M., Siebes, R., Staab, S., and Stuckenschmidt, H. (2003). A metadata model for semantics-based peer-to-peer systems. In *Proceedings of the second Konferenz Professionelles Wissensmanagement*, Lucern.

[Ellman, 1989] Ellman, T. (1989). Explanation-based learning: a survey of programs and perspectives. *ACM Comput. Surv.*, 21(2):163–221.

[Esteva et al., 2001] Esteva, M., Rodriguez, J. A., Sierra, C., Garcia, P., and Arcos, J. L. (2001). On the formal specifications of electronic institutions. In *Agent-mediated Electronic commerce (The European AgentLink Perspective)*, pages 126–147. LNAI.

[Farquhar et al., 1996] Farquhar, A., Fikes, R., and Rice, J. (1996). The Ontolingua server: A tool for collaborative ontology construction. Technical Report 96-26, Stanford KSL.

[Fensel et al., 2001] Fensel, D., Horrocks, I., van Harmelen, F., McGuinness, D. L., and Patel-Schneider, P. F. (2001). OIL: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–45.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 2(3/4), 189–208.

[Finkelstein et al., 1993] Finkelstein, A., Gabbay, D. M., Hunter, A., Kramer, J., and Nuseibeh, B. (1993). Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, pages 84–99.

[FIPA, 1997] FIPA (1997). Fipa 97 specification, version 2.0, part 2. http://www.fipa.org/spec/FIPA97.html.

[FIPA, 2005] FIPA (2005). http://www.fipa.org/.

[FIPA-ACL, 2002] FIPA-ACL (2002). http://www.fipa.org/specs/fipa00061/xc00061e.html.

[Fox and Long, 2003] Fox, M. and Long, D. (2003). PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of AI Research*, 20:61–124.

[Frame Ontology, 1999] Frame Ontology (1999). Ontolingua frame ontology. http://www-sop.inria.fr/acacia/personnel/phmartin/RDF/frameOntology.html.

[Franconi et al., 2000] Franconi, E., Grandi, F., and Mandreoli, F. (2000). A semantic approach for schema evolution and versioning in object-oriented databases. *Lecture Notes in Computer Science*, 1861:1048–1062.

[Franklin and Graesser, 1997] Franklin, S. and Graesser, A. (1997). Is it an agent, or just a program?: A taxonomy for autonomous agents. In *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*, pages 21–35, London, UK. Springer-Verlag.

[Frańová and Kodratoff, 1992] Frańová, M. and Kodratoff, Y. (1992). Predicate synthesis from formal specifications. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 87–91, New York, NY, USA. John Wiley & Sons, Inc.

[Gärdenfors, 1992] Gärdenfors, P. (1992). Belief revision: An introduction. In Gärdenfors, P., editor, *Belief Revision*, pages 1–28. Cambridge University Press. Cambridge Tracts in Theoretical Computer Science.

[Gärdenfors and Rott, 1995] Gärdenfors, P. and Rott, H. (1995). Belief revision. In Gabbay, D., Hogger, C. J., and Robinson, J., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4, pages 35–132. Oxford Science Publications.

[Genesereth and Fikes, 1992] Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Stanford, CA, USA.

[Gerevini et al., 2003] Gerevini, A., Saetti, A., and Serina, I. (2003). Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research (JAIR)*, 20:239–290.

[Giunchiglia, 1993] Giunchiglia, F. (1993). Contextual reasoning. *Epistemologia - Special Issue on "I Linguaggi e le Macchine"*, XVI:345–364.

[Giunchiglia and Shvaiko, 2003] Giunchiglia, F. and Shvaiko, P. (2003). Semantic matching. In Giunchiglia, F., Gomez-perez, A., Pease, A., Stuckenschmidt, H., Sure, Y., and Willmott, S., editors, *Workshop on ontologies and distributed systems (ODS 2003)*, Acapulco.

[Giunchiglia et al., 2005a] Giunchiglia, F., Shvaiko, P., and Yatskevich, M. (2005a). S-match: an algorithm and an implementation of semantic matching. In Kalfoglou, Y., Schorlemmer, M., Sheth, A., Staab, S., and Uschold, M., editors, *Semantic Interoperability and Integration*, number 04391 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany.

[Giunchiglia and Walsh, 1990] Giunchiglia, F. and Walsh, T. (1990). The use of abstraction in automatic inference. *Proceedings of UK Conference on Information Technology (IT-90)*.

[Giunchiglia and Walsh, 1992] Giunchiglia, F. and Walsh, T. (1992). A theory of abstraction. *Artificial Intelligence*, 56.

[Giunchiglia et al., 2005b] Giunchiglia, F., Yatskevich, M., and Giunchiglia, E. (2005b). Efficient semantic matching. In perez A., G. and J., E., editors, *Proceedings of the 2nd European semantic web conference (ESWC'05)*, volume 3532/2005 of *Lecture Notes in Computer Science*, Heraklion, Crete, Greece.

[Gruber, 1991] Gruber, T. R. (1991). The role of common ontology in achieving sharable, reusable knowledge bases. In Allen, J., Fikes, R., and Sandewall, E., ed-

itors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second International Conference (KR'91)*, pages 601–602. Kaufmann, San Mateo, CA.

[Gruber, 1992] Gruber, T. R. (1992). Ontolingua: A mechanism to support portable ontologies. Technical report, Knowledge Systems Laboratory, Stanford University, Stanford, CA.

[Gruber, 1993] Gruber, T. R. (1993). A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2).

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing. In *International Journal of Human-Computer Studies*, volume 43(5-6), pages 907–928.

[Guha and Hayes, 2003] Guha, R. V. and Hayes, P. (2003). LBase: Semantics for languages of the Semantic Web, W3C Working Group Note. http://www.w3.org/TR/lbase.

[Haase et al., 2004] Haase, P., Siebes, R., and van Harmelen, F. (2004). Peer selection in peer-to-peer networks with semantic topologies. In Bouzeghoub, M., editor, *Proceedings of the International Conference on Semantics in a Networked World (ICNSW'04)*, volume 3226 of *LNCS*, pages 108–125, Paris. Springer Verlag.

[Hameed et al., 2003] Hameed, A., Preece, A., and Sleeman, D. (2003). Ontology reconciliation. In Staab, S. and Studer, R., editors, *Handbook on Ontologies in Information Systems*, pages 231 – 250. Springer Verlag, Germany.

[Hameed et al., 2002] Hameed, A., Sleeman, D., and Preece, A. (2002). Detecting mismatches among experts; ontologies acquired through knowledge elicitation. *Knowledge-Based Systems*, 15(05-Jun):265 – 273.

[Hayes and Menzel, 2001] Hayes, P. and Menzel, C. (2001). A semantics for the knowledge interchange format. In *IJCAI 2001 Workshop on the IEEE Standard Upper Ontology*.

[Hayes-Roth, 1983] Hayes-Roth, F. (1983). Using proofs and refutations to learn from experience. In *Machine Learning*, pages 221–240. Tioga Publishing, Palo Alto, CA.

[Heflin and Hendler, 2000] Heflin, J. and Hendler, J. A. (2000). Dynamic ontologies on the web. In *AAAI/IAAI*, pages 443–449.

[Hoffmann, 2002] Hoffmann, J. (2002). Extending FF to numerical state variables. In Harmelen, F. V., editor, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 571–575, Lyon, France.

[Hoffmann, 2003] Hoffmann, J. (2003). The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341.

[Hoffmann, 2005] Hoffmann, J. (2005). FF. http://www.mpi-sb.mpg.de/ hoffmann/ff.html.

[Hoffmann and Nebel, 2001] Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[Horrocks and Patel-Schneider, 2003] Horrocks, I. and Patel-Schneider, P. F. (2003). Three theses of representation in the semantic web. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 39–47, New York, NY, USA. ACM Press.

[Huang et al., 2005] Huang, Z., van Harmelen, F., and ten Teije, A. (2005). Reasoning with inconsistent ontologies. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, Edinburgh, Scotland.

[Huhns and Stephens, 1999] Huhns, M. N. and Stephens, L. M. (1999). Multiagent systems and societies of agents. In Weiß, G., editor, *A Modern Approach to Distributed Artificial Intelligence*, chapter 2. The MIT Press, San Francisco, CA.

[Hutter, 2000] Hutter, D. (2000). Management of change in structured verification. In *Proceedings of 15th IEEE International Conference on Automated Software Engineering, ASE-2000*, pages 23–24. IEEE Computer Society.

[Jennings, 1999] Jennings, N. (1999). Agent-oriented software engineering. In *Proceedings of the 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence*, pages 4–10, Cairo, Egypt. (Invited paper).

[Kalfoglou, 2002] Kalfoglou, Y. (2002). *Exploring Ontologies*, pages 863–887. Handbook of Software Engineering and Knowledge Engineering: vol. 1: Fundamentals. World Scientific Publishing.

[Kalfoglou and Schorlemmer, 2003a] Kalfoglou, Y. and Schorlemmer, M. (2003a). If-map: an ontology mapping method based on information flow theory. *Journal on Data Semantics*, 1(1):98–127.

[Kalfoglou and Schorlemmer, 2003b] Kalfoglou, Y. and Schorlemmer, M. (2003b). Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18:1:1–31.

[Kalfoglou and Schorlemmer, 2004] Kalfoglou, Y. and Schorlemmer, M. (2004). Formal support for representing and automating semantic interoperability. In *1st European Semantic Web Symposium (ESWS'04), Springer LNCS 3053, Heraklion, Crete, Greece*, pages 45–61.

[Kautz and Selman, 1999] Kautz, H. and Selman, B. (1999). Unifying SAT-based and graph-based planning. In Minker, J., editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland. Computer Science Department, University of Maryland.

[Kautz and Selman, 1992] Kautz, H. A. and Selman, B. (1992). Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363.

[Kent, 2000] Kent, R. E. (2000). The information flow foundation for conceptual knowledge organization. In *Proceedings of the 6th International Conference of the International Society for Knowledge Organization (IKSO).*

[Kifer and Lausen, 1989] Kifer, M. and Lausen, G. (1989). F-logic: A higher-order language for reasoning about objects, inheritance and scheme. In *Proceedings of ACM SIGMOD Conference on Management of Data*, volume 18, pages 134–146, Portland, Oregan.

[Klein, 2002] Klein, M. (2002). Ontology versioning and change detection on the web. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Siguenza, Spain.

[Koivunen and Miller, 2001] Koivunen, M.-R. and Miller, E. (2001). W3C Semantic Web activity. In *Proceedings of the Semantic Web Kick-Off Seminar in Finland.*

[Koppel et al., 1994] Koppel, M., Feldman, R., and Segre, A. M. (1994). Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208.

[KQML, 2005] KQML (2005). http://www.cs.umbc.edu/kqml/papers/kbkshtml/kbks.html.

[Lakatos, 1976] Lakatos, I. (1976). *Proofs and Refutations*. Cambridge University Press.

[Lassila and Swick, 1999] Lassila, O. and Swick, R. (1999). Resource description framework (RDF) model and syntax specification. W3C Recommendation, World Wide Web Consortium, http://www.w3.org/TR/REC-rdf-syntax/.

[Luger and Stubblefield, 1997] Luger, G. and Stubblefield, W. (1997). *Artificial Intelligence: Structure and Strategies for Complex Problem Solving*. Addison-Wesley.

[Mauthe and Hutchison, 2003] Mauthe, A. and Hutchison, D. (2003). Peer-to-Peer Computing: Systems, Concepts and Characteristics. *Praxis in der Informationsverarbeitung & Kommunikation (PIK), K. G. Sauer Verlag, Special Issue on Peer-to-Peer*, 26(03/03).

[McCarthy and Hayes, 1969] McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B. and Michie, D., editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press. reprinted in McC90.

[McGinnis and Robertson, 2004a] McGinnis, J. and Robertson, D. (2004a). Dynamic and distributed interaction protocols. In *Adaptive Agents and MAS II*, volume 3394-0178 of *LNAI*. Springer-Verlag.

[McGinnis and Robertson, 2004b] McGinnis, J. and Robertson, D. (2004b). Realizing agent dialogues with distributed protocols. In *Developments in Agent Communication*, volume 3396 of *LNAI*. Springer-Verlag.

[McGuinness et al., 2000] McGuinness, D. L., Fikes, R., Rice, J., and Wilder, S. (2000). An environment for merging and testing large ontologies. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning (KR2000)*, pages 483–493, Colorado, USA.

[McIlraith et al., 2001] McIlraith, S., Son, T., and Zeng, H. (2001). Semantic Web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2):46–53. citeseer.nj.nec.com/mcilraith01semantic.html.

[McNeill et al., 2003a] McNeill, F., Bundy, A., and Schorlemmer, M. (2003a). Dynamic ontology refinement. In *Proceedings of ICAPS'03 Workshop on Plan Execution*, Trento, Italy.

[McNeill et al., 2004a] McNeill, F., Bundy, A., and Walton, C. (2004a). An automatic translator from KIF to PDDL. In *Proceedings of the 23rd Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2004)*, Cork, Ireland.

[McNeill et al., 2004b] McNeill, F., Bundy, A., and Walton, C. (2004b). Diagnosing and repairing ontological mismatches. In *Proceedings of the second starting AI Researchers' symposium*, Valencia, Spain.

[McNeill et al., 2004c] McNeill, F., Bundy, A., and Walton, C. (2004c). Facilitating agent communication through detecting, diagnosing and refining ontological mismatch. In *Proceedings of the KR2004 Doctoral Consortium*. AAAI Technical Report.

[McNeill et al., 2005] McNeill, F., Bundy, A., and Walton, C. (2005). Planning from rich ontologies through translation betweeen representations. In *Proceedings of ICAPS'05 Workshop on The Role of Ontologies in Planning and Scheduling*, Monterey, CA, USA.

[McNeill et al., 2003b] McNeill, F., Bundy, A., Walton, C., and Schorlemmer, M. (2003b). Plan execution failure analysis using plan deconstruction. In *Proceedings of the 22nd Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2003)*.

[Minsky, 1985] Minsky, M. (1985). A framework for representing knowledge. In Brachman, R. J. and Levesque, H. J., editors, *Readings in Knowledge Representation*, pages 245–262. Kaufmann, Los Altos, CA.

[Mitchell et al., 1986] Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Mach. Learn.*, 1(1):47–80.

[Mitra et al., 2000] Mitra, P., Wiederhold, G., and Kersten, M. (2000). A graph-oriented model for articulation of ontology interdependencies. *Lecture Notes in Computer Science*, 1777:86–100.

[Monroy, 1999] Monroy, R. (1999). The use of proof planning failure for correcting faulty conjectures. In Ahuactzin, J., editor, *Encuentro Nacional de Cumputacion, Taller de Logica*, volume Pachuca, pages 1–6. Socidad Mexicana de Ciencia de la Computacion.

[Monroy and Bundy, 2001] Monroy, R. and Bundy, A. (2001). On the correction of faulty formulae. *Computacin y Sistemas*, pages 5(1):25–37.

[Mooney, 1995] Mooney, R. J. (1995). *A preliminary PAC analysis of theory revision*, volume III: Selecting Good Models, pages 43–53. MIT Press.

[Moore, 1974] Moore, J. (1974). *Computational Logic: Structure sharing and proof of program properties, part II*. PhD thesis, University of Edinburgh. Available from Edinburgh as DCL memo no. 68 and from Xerox PARC, Palo Alto as CSL 75-2.

[Muggleton, 1999] Muggleton, S. (1999). Inductive Logic Programming. In *The MIT Encyclopedia of the Cognitive Sciences (MITECS)*. MIT Press.

[Muggleton and Feng, 1990] Muggleton, S. and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan.

[Narayanan and McIlraith, 2002] Narayanan, S. and McIlraith, S. (2002). Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*.

[Nardi and Brachman, 2003] Nardi, D. and Brachman, R. J. (2003). An introduction to description logics. In Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 1–40. Cambridge University Press.

[NESC, 2005] NESC (2005). http://www.nesc.ac.uk/index.html.

[Nisbett and Wilson, 1984] Nisbett, R. E. and Wilson, T. D. (1984). Telling more than we can know: Verbal reports on mental processes. *Psychological Review*, pages 231–259.

[Noy and Musen, 2000] Noy, N. F. and Musen, M. A. (2000). PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 450–455, Austin, TX.

[Noy and Musen, 2001] Noy, N. F. and Musen, M. A. (2001). Anchor-prompt: Using non-local context for semantic matching. In *Workshop on Ontologies and Information Sharing at the seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, WA.

[Noy and Musen, 2002] Noy, N. F. and Musen, M. A. (2002). Promptdiff: a fixed-point algorithm for comparing ontology versions. In *Eighteenth national conference on Artificial intelligence*, pages 744–750, Edmonton, Alberta, Canada. American Association for Artificial Intelligence.

[Oriol, 2003] Oriol, M. (2003). Peer services: from description to invocation. In *Proceedings of the International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2002)*, volume 2530 of *LNCS*, pages 21–32. Springer-Verlag Heidelberg.

[Ourston and Mooney, 1990] Ourston, D. and Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. *Proceedings of the National Conference on Artificial Intelligence*, 2:815–820.

[Parsia et al., 2005] Parsia, B., Sirin, E., and Kalyanpur, A. (2005). Debugging owl ontologies. In *The 14th International World Wide Web Conference (WWW2005)*, Chiba, Japan.

[Patterson et al., 1993] Patterson, L. I., Turner, R. S., and Hyatt, R. M. (1993). Construction of a fault-tolerant distributed tuple-space. In *Proceedings of the 1993*

ACM/SIGAPP symposium on Appliced computing: states of the art and practice, Indianapolis, United States.

[Pease et al., 2004] Pease, A., Colton, S., Smaill, A., and Lee, J. (2004). A model of Lakatos's philosophy of mathematics. In *Computing and Philosophy (ECAP)*.

[PlanComp2, 2000] PlanComp2 (2000). Second international planning competition. http://www.cs.toronto.edu/aips2000/.

[PlanComp3, 2002] PlanComp3 (2002). Third international planning competition. http://planning.cis.strath.ac.uk/competition/.

[PLANET, 2004] PLANET (2004). http://scom.hud.ac.uk/planet/repository/.

[Protzen, 1996] Protzen, M. (1996). Patching faulty conjectures. *LNCS*, 1104:77–91.

[PSL, 2005] PSL (2005). PSL. http://www.mel.nist.gov/psl/.

[Quillian, 1967] Quillian, M. (1967). Word Concepts: A theory and simulation of some basic capabilities. *Behavioral Science*, 12:410–430.

[RCUK, 2004] RCUK (2004). Research councils UK. http://www.rcuk.ac.uk/escience/.

[Rector, 2002] Rector, A. (2002). Normalisation of ontology implementations: Towards modularity, re-use, and maintainability. In *Proceedings of the Workshop on Ontologies for Multiagent Systems (OMAS) in conjunction with the European Knowledge Acquisition Workshop*, Siguenza, Spain.

[Rector et al., 2001] Rector, A. L., Wroe, C., Rogers, J., and Roberts, A. (2001). Untangling taxonomies and relationships: personal and practical problems in loosely coupled development of large ontologies. In *K-CAP*, pages 139–146.

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95.

[Richards and Mooney, 1995] Richards, B. L. and Mooney, R. J. (1995). Automated refinement of first-order Horn-clause domain theories. *Machine Learning, 19, 2*, pages 95–131.

[Russell and Norvig, 1995] Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall International Editions.

[Scholbach, 2005] Scholbach, S. (2005). Diagnosing terminologies. In *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI'05)*, pages 670–675, Pittsburgh, Pennsylvania.

[Scholbach and Cornet, 2003] Scholbach, S. and Cornet, R. (2003). Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of the eighteenth International Joint Conference in Artificial Intelligence (IJCAI'03)*, pages 1393–1400.

[Schorlemmer et al., 2002] Schorlemmer, W., Potter, S., Robertson, D., and Sleeman, D. (2002). Formal knowledge management in distributed environments. *Workshop on Knowledge Transformations for the Semantic Web, Europeran Conference of Artificial Intelligence (ECAI-02)*.

[Searle, 1970] Searle, J. R. (1970). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press.

[SemanticWeb.org, 2002] SemanticWeb.org (2002). http://www.semanticweb.org.

[Services, 2005] Services, O. (2005). http://www.daml.org/services/owl-s/1.0/.

[Shapiro, 1982] Shapiro, E. Y. (1982). *Algorithmic Program Debugging*. The MIT Press.

[Shaw and Gaines, 1989] Shaw, M. and Gaines, B. (1989). Comparing conceptual structures: Consensus, conflict, correspondence and contrast. *Knowledge Acquisition*, 1:341–363.

[Sicstus, 2005] Sicstus (2005). Sicstus user's manual. http://www.sics.se/sicstus/docs/latest/html/sicstus/.

[Siebes, 2005] Siebes, R. (2005). pnear: combining content clustering and distributed hash tables. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Knowledge Management (P2PKM)*, San Diego, CA, USA.

[Sierra et al., 1998] Sierra, C., Jennings, N. R., Noriega, P., and Parsons, S. (1998). A framework for argumentation-based negotiation. In Singh, M. P., Rao, A., and Wooldridge, M. J., editors, *Fourth International Workshop on Agent Theories Architectures and Languages (ATAL-97)*, volume 1365 of *Lecture Notes in Computer Science*, pages 177–192. Springer-Verlag.

[Singh, 2002] Singh, M. P. (2002). Peer-to-peer computing for information systems. In *AP2PC*, pages 15–20.

[Sleeman et al., 2002] Sleeman, D., Potter, S., Robertson, D., and Schorlemmer, W. (2002). Enabling services for distributed environments: ontology extraction and knowledge base characterisation. *ECAI Workshop on Knowledge Transformations for the Semantic Web*.

[Sloan and Turán, 1999] Sloan, R. H. and Turán, G. (1999). On theory revision with queries. In *Proc. 12th Annu. Conf. on Comput. Learning Theory*, pages 41–52. ACM Press, New York, NY.

[Stuckenschmidt, 2002] Stuckenschmidt, H. (2002). Exploiting partially shared ontologies for multi-agent communication. In *CIA '02: Proceedings of the 6th International Workshop on Cooperative Information Agents VI*, pages 249–263, London, UK. Springer-Verlag.

[Stuckenschmidt et al., 2005] Stuckenschmidt, H., Giunchiglia, F., and van Harmelen, F. (2005). Query processing in ontology-based peer-to-peer systems. In Tamma, V., Cranefield, S., Finin, T., and Willmott, S., editors, *Ontologies for Agents: Theory and Experiences*, Whitestein Series in Software Agent Technologies. Birkhuser.

[Stuckenschmidt and Klein, 2003] Stuckenschmidt, H. and Klein, M. (2003). Integrity and change in modular ontologies. In *International Joint Conference on Artificial Intelligence - IJCAI '03*, pages 900–905, Acapulco, Mexico.

[SUMO, 2005] SUMO (2005). SUMO. http://ontology.teknowledge.com/.

[SUO-KIF, 2003] SUO-KIF (2003). http://suo.ieee.org/SUO/KIF/suo-kif.html.

[SWSI, 2005] SWSI (2005). http://www.swsi.org/.

[Uschold, 2002] Uschold, M. (2002). Creating semantically integrated communities on the world wide web. In *WWW'02 Semantic Web Workshop*.

[van Harmelen, 2002] van Harmelen, F. (2002). The complexity of the web ontology language. *IEEE Intelligent Systems*, 17(2):71–72.

[van Harmelen, 2004] van Harmelen, F. (2004). The semantic web: What, why, how, and when. *IEEE Distributed Systems Online*, 5(3).

[Vasconcelos, 2002] Vasconcelos, W. (2002). Skeleton-based agent development for electronic institutions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. ACM Press. Part II.

[Visser et al., 1997] Visser, P. R. S., Jones, D. M., Bench-Capon, T. J. M., and Shave, M. J. R. (1997). An analysis of ontological mismatches: Heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA.

[W3C, 2005] W3C (2005). http://www.w3.org/2001/sw/.

[Wache et al., 2001] Wache, H., Vögele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., and Hübner, S. (2001). Ontology-based integration of information — a survey of existing approaches. In Stuckenschmidt, H., editor, *IJCAI–01 Workshop: Ontologies and Information Sharing*, pages 108–117.

[Walton, 2005] Walton, C. (2005). Typed protocols for peer-to-peer service composition. In *Proceedings of the Second International Workshop on Peer-to-Peer Knowledge Management (P2PKM 2005)*, San Deigo, USA.

[Weiß, 1999] Weiß, G. (1999). Introduction to multiagent systems. In Weiß, G., editor, *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 1–9. MIT Press.

[Whittle et al., 1997] Whittle, J., Bundy, A., and Lowe, H. (1997). An editor for helping novices to learn Standard ML. *Proceedings of the Ninth International Symposium on Programming Languages, Implementations, Logics and Programs*, pages 389–405.

[Wiederhold, 1997] Wiederhold, G. (1997). Mediators in the architecture of future information systems. In Huhns, M. N. and Singh, M. P., editors, *Readings in Agents*, pages 185–196. Morgan Kaufmann, San Francisco, CA, USA.

[Wiesman et al., 2002] Wiesman, F., Roos, N., and Vogt, P. (2002). Automatic ontology mapping for agent communication. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 563–564, New York, NY, USA. ACM Press.

[Wooldridge, 1999] Wooldridge, M. (1999). Intelligent agents. In Weiß, G., editor, *A Modern Approach to Distributed Artificial Intelligence*, chapter 3. The MIT Press, San Francisco, CA.

[Wooldridge, 2002] Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley and Sons.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152.

[WSMO, 2005] WSMO (2005). Web service modeling ontology. http://www.wsmo.org/.