THE UNIVERSITY OF NOTTINGHAM

Truth Maintenance in Knowledge-Based Systems

by

Hai Hoang Nguyen

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

April 2014

Abstract

Truth Maintenance Systems (TMS) have been applied in a wide range of domains, from diagnosing electric circuits to belief revision in agent systems. There also has been work on using the TMS in modern Knowledge-Based Systems such as intelligent agents and ontologies. This thesis investigates the applications of TMSs in such systems.

For intelligent agents, we use a "light-weight" TMS to support query caching in agent programs. The TMS keeps track of the dependencies between a query and the facts used to derive it so that when the agent updates its database, only affected queries are invalidated and removed from the cache. The TMS employed here is "light-weight" as it does not maintain all intermediate reasoning results. Therefore, it is able to reduce memory consumption and to improve performance in a dynamic setting such as in multi-agent systems.

For ontologies, this work extends the Assumption-based Truth Maintenance System (ATMS) to tackle the problem of axiom pinpointing and debugging in ontology-based systems with different levels of expressivity. Starting with finding all errors in auto-generated ontology mappings using a "classic" ATMS [23], we extend the ATMS to solve the axiom pinpointing problem in Description Logics-based Ontologies. We also attempt this approach to solve the axiom pinpointing problem in a more expressive upper ontology, SUMO, whose underlying logic is undecidable.

Acknowledgements

I would like to take this opportunity to thank my first supervisor, Dr. Natasha Alechina, for her support, encouragement, and great patience during my time in Nottingham, in both undergraduate and PhD level. Without her help, it would not have been possible for me to write and complete this thesis.

I am grateful to Dr. Brian Logan, my second supervisor, for introducing me Intelligent Agents and Truth Maintenance Systems, and for his thorough and insightful comments during my PhD research. His G53DIA module is the best module I have ever taken.

I would like to thank Professor Tony Pridmore and Dr. Jeff Z. Pan for being my examiners and for the constructive comments on the first version of this thesis.

I would also like to thank my colleagues from the Agent Labs (it's a pleasure to work in C50): Nga, Trang, Julian, Susan and my best friend in Nottingham, Khin Lwin, for friendship, support, and discussions during my time in Nottingham.

I would not be able to go that far without my parents. I am grateful to them for supporting my education.

Finally, I would like to thank my wife, Phuong Dang, a.k.a. my "motivation maintenance system", for everything.

Contents

Abstract **i**

Acknowledgements **ii**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**
- 1.1 Key Concepts 2
- 1.2 Motivation 3
- 1.3 Research Objectives 5
- 1.4 Structure of the Thesis 5

2 Intelligent Agents and Knowledge-based Systems **8**
- 2.1 Introduction 8
- 2.2 Intelligent Agents 9
  - 2.2.1 Agents and the Environments 9
  - 2.2.2 BDI Model 10
  - 2.2.3 Agent Programming Languages 12
  - 2.2.4 The GOAL Agent Programing Language 13
- 2.3 Knowledge-based Systems 15
- 2.4 Conclusion 17

3 Description Logics and Ontology Debugging **18**
- 3.1 Introduction 18
- 3.2 Description Logics - An Overview 20
  - 3.2.1 $\mathcal{ALC}$ Syntax and Semantics 20
  - 3.2.2 TBox and ABox 23
  - 3.2.3 Reasoning Tasks for a DL-based KB 25
  - 3.2.4 Tableau-based Reasoning 26
- 3.3 Axiom Pinpointing and Debugging in a DL-based Ontology . . . 27
  - 3.3.1 Axiom Pinpointing 27

3.3.2 Ontology Debugging . . . 29
3.4 Related Work on Ontology Debugging . . . 32
3.4.1 Black-box Approach . . . 32
3.4.2 Glass-box Approach . . . 35
3.5 Conclusion . . . 37

4 Truth Maintenance Systems **38**
4.1 Introduction . . . 38
4.2 Data Structures in a TMS . . . 40
4.3 JTMS and ATMS: The Differences . . . 42
4.4 Justification-based Truth Maintenance Systems . . . 44
4.4.1 Data Structures in a JTMS . . . 44
4.4.2 Main Operations of a JTMS . . . 45
4.4.3 Representing Negation and Disjunction in a JTMS . . . 47
4.5 Assumption-based Truth Maintenance Systems . . . 48
4.5.1 Structure of an ATMS node . . . 48
4.5.2 Properties of ATMS node labels . . . 50
4.5.3 Label Update Propagation Algorithms . . . 52
4.5.4 Implementing Disjunctions in an ATMS . . . 52
4.6 Applications of Truth Maintenance Systems in Knowledge-Based Systems . . . 54
4.7 Conclusion . . . 56

5 Query Caching in Agent Programs **57**
5.1 Introduction . . . 57
5.2 Queries and Updates in Agent Reasoning Cycles . . . 59
5.2.1 Queries . . . 60
5.2.2 Updates . . . 61
5.2.3 Agent Query-Update Cycles . . . 63
5.3 Query Caching Modes: Single-Cycle vs. Multi-Cycle . . . 67
5.3.1 The Abstract Performance Model of Query Caching . . . 67
5.3.2 Single-Cycle Query Caching . . . 68
5.3.3 Multi-Cycle Query Caching . . . 70
5.4 Truth Maintenance for Multi-Cycle Query Caching . . . 74
5.5 Experimental Results . . . 78
5.6 Related Work . . . 80
5.7 Conclusion . . . 81

6 Detecting Geospatial Ontology Mapping Errors **82**
6.1 Introduction . . . 82
6.2 Finding Incorrect Mappings in a Geospatial Knowedge-based System using the Logic of NEAR and FAR . . . 84
6.3 The ATMS-based Approach to Mapping Errors-Detection . . . 87
6.4 Algorithms for Label Update Propagation in the ATMS . . . 88

6.5 Correctness 91
6.6 Preliminary Experimental Results 93
6.7 Conclusion 96

7 Debugging Ontologies with Disjunctions and Loops 98
7.1 Introduction 98
7.2 System Architecture 100
7.3 The Reasoner 102
7.3.1 A Dictionary $\mathcal{ALC}$ 102
7.3.2 Loops and The Blocking Conditions 104
7.3.2.1 Blocking Condition without Pinpointing 106
7.3.2.2 Blocking Condition with Pinpointing 107
7.4 The D-ATMS 109
7.4.1 Dealing with Disjunctions 110
7.4.2 The D-ATMS Data-structures: Nodes, Justifications, Environments and Labels 112
7.4.3 Example 114
7.4.4 Label Computation 116
7.4.5 Lazy Label Update Propagation 118
7.5 Correctness 123
7.6 Experimental Results 128
7.7 Displaying Explanations 133
7.8 Conclusion 134

8 Axiom Pinpointing for SUMO 136
8.1 Introduction 136
8.2 SUMO & SUO-KIF 139
8.3 FKIF 141
8.4 Axiom Pinpointing for SUMO 145
8.4.1 Reasoner 146
8.4.2 Truth Maintenance System 148
8.5 Examples 151
8.6 Related work 153
8.7 Conclusion 153

9 Conclusion and Future Work 155
9.1 Summary of Contributions 155
9.2 Future Work 157

Bibliography 160

List of Figures

2.1 An agent and its environment. This figure is based on the one in [79]. 10
2.2 The Procedural Reasoning System (see [40]). 11

3.1 A simple ontology describing animal concepts. 19

4.1 Communication between a TMS and the reasoner in a problem solver. 39
4.2 An example of the graph of datum nodes and justifications. Diamond and circles are justifications and nodes in the dependency network. 41
4.3 Labeling node $n1$ from *out* to *in*. White and dark-colour nodes represent *out* nodes and *in* respectively. Diamond and circles are justifications and nodes in the dependency network. 46
4.4 The ATMS operates in a cycle. 53

5.1 An example of query-update cycle in the BW agent 65
5.2 A simple search tree from query $son(haran, X)$ from [87]. 75
5.3 Mappings from queries to their support sets and from facts to their invalidated sets. 76
5.4 Prolog queries and facts in correspondence to TMS datum nodes and assumptions. 77

6.1 The Framework to Find Incorrect Mappings of 2 Geospatial Ontologies, Open Street Map (OSM) and Ordnance Survey Great Britain (OSGB). 88

7.1 The components of AOD: a reasoner and the D-ATMS 100
7.2 Tableau with nested disjunctions 104
7.3 Tableau reasoning with disjunctions 110
7.4 The inference from a conjunction and a disjunction might be recored by a typical ATMS 111
7.5 Justification graph. Formula nodes are round, axioms are blue, $\perp$ is red. Justification nodes are square, non-deterministic justifications are green with dashed arrows. 116

8.1 BNF syntax for SUO-KIF 140

List of Tables

5.1 Comparison of different caching modes 80

7.1 Execution times (in ms) for the Biochemistry-primitive ontology (265 axioms and 10 unsatisifiable concepts). 130
7.2 Execution times (in ms) for the BuildingsAndPlaces ontology (124 axioms and 10 unsatisifiable concepts). 130
7.3 Execution times (in ms) for the Adult Mouse Brain ontology (3447 axioms and 10 unsatisifiable concepts). 130
7.4 Execution times (in ms) for the Geo ontology (500 axioms and 11 unsatisifiable concepts). 132
7.5 Execution times (in ms) for the DICE ontology (27939 axioms and 76 unsatisifiable concepts). 132
7.6 Execution times (in ms) for the MGED ontology (406 axioms and 72 unsatisifiable concepts). 132
7.7 Execution times (in ms) for the Mini Sumo & Cyc ontology (5725 axioms and 923 unsatisifiable concepts). 132

List of Algorithms

3.1 Find all explanations for a concept unsatisfiability given a function to find a single explanation 34
5.1 Multi-Cycle Caching 70
6.1 Propagate incremental label update 89
6.2 Compute single label update 89
6.3 Update the current label and propagate to consequences 90
6.4 Update nodes when a nogood e is found 91
7.1 Update a non-deterministic justification 120
7.2 Update a deterministic justification 121
7.3 Update the label of a node and propagate to consequents 122
7.4 Remove nogood environments from node labels 123

Chapter 1

Introduction

Rational agents, including humans, need some form of knowledge. Knowledge is the ability of an agent to represent what she knows or believes about the world (representation) and to infer new knowledge from her current beliefs and knowledge (reasoning). Elements of knowledge are, therefore, not separate but instead connected to each other via some reasoning procedures. Some elements of knowledge are assumptions, i.e., they are assumed explicitly by the agent, while others are derivable from assumptions by reasoning processes. Because assumptions can come from different sources, knowledge inconsistency is unavoidable. To understand why knowledge is inconsistent, it is necessary to keep track of which pieces of knowledge derive another piece. This dependency tracking technique is referred to *"truth maintenance"*,[1] or more precisely reason maintenance systems. This thesis is the investigation of the use of truth maintenance approaches in knowledge-based systems, especially in ontology systems and intelligent agent systems.

[1]It is interesting that although the main topic of TMS is to maintain consistency, the term "truth maintenance" itself has not been used consistently. The term "truth maintenance systems" was firstly introduced by Doyle [30]. However, it is later referred to a more precise term "reason maintenance systems" by Nebel in [65]. In fact, what a TMS aims to maintain is the reasons to keep a belief, not the truth. However, as "truth maintenance systems" have been used widely and persistently in the AI literature, we use this term in the rest of this thesis for consistency.

In the rest of this chapter, we present the key-concepts covered in this thesis, the motivation of the work, its aims and objectives, and the structure of the thesis.

1.1 Key Concepts

For readability, we list below the key concepts which are frequently mentioned in the rest of this thesis.

Knowledge-based systems are systems which are able to *represent* knowledge and to *exploit* its knowledge to solve particular tasks using reasoning procedures.

Intelligent agents are entities which perceive their environment through sensors and act through actuators to achieve some goals or to perform some particular tasks.

Ontologies are knowledge bases which can represent knowledge in term of concepts, instances, and their relationships. Nowadays ontologies are usually referred as Description Logic-based knowledge bases.

An upper ontology defines abstract concepts which will then be used by domain specific ontologies to define more concrete concepts for different applications.

Ontology mapping is the process of mapping from an entity in one ontology to one in another ontology. The mapping can be done in concept-level, where a concept is mapped onto another concept, or in instance-level, where an individual is mapped onto another individual. Ontology mapping aims to overcome heterogeneity among ontologies.

1.2 Motivation

Truth Maintenance Systems (TMS) have been an active research area whose applications range from physical domains to software systems. Notable applications of TMS include model-based diagnosis (i.e., generating diagnoses or explanations for system faults) and belief revision (i.e., revising the knowledge base to adapt new knowledge updates). In general, these applications are based on the ability to keep track of dependencies between data of a TMS. In my undergraduate dissertation, I had a chance to work on one type of TMS, the Logic-based TMS, to solve the problem of belief revision in an agent's belief base [68]. The TMS is proved to be a very powerful tool and can be employed in different problems and domains. A question has arised: how to construct different types of TMS to solve problems in modern knowledge-based systems such as ontologies in the Semantic Web and the knowledge bases of intelligent agents. Surprisingly, there has not been many works on the applications of TMS in such systems, especially in the Semantic Web area.

Therefore, this thesis aims to address the following specific research questions:

Research questions *Which applications could the approach taken by Truth Maintenance Systems, i.e., maintaining a dependency graph of data, deliver for intelligent agent systems and ontology systems? How could a TMS be constructed to provide such applications in these systems?*

An intelligent agent uses a knowledge-based system to represent and reason about its knowledge. In particular, an agent program sends queries to the KBS and receives answers. The answer to such a query is based on whether the reasoner can derive an instance of the query using the agent's current beliefs. Such queries to the reasoner may be costly, especially with loosely-coupled reasoners which require a third-party interface. For example, SWI-Prolog [88] needs a

Prolog/Java interface layer[2] to be used in a Java-based agent platform such as the GOAL agent programming language [48]. Therefore, caching such queries might potentially bring benefits to the performance of an agent program. However, updates in the agent databases can invalidate cached results. As a result, a question to answer is that how can the agent know which cached results are affected by an update. Obviously, reasoner inferences can be recorded using the justifications in a TMS dependency graph, and hence a TMS can be used to answer this question. In fact, Truth Maintenance Systems have also been used in many agents and multi-agent systems [3, 52, 61]. However, previous works only use TMS on handling inconsistency in the agent's belief base,e.g., belief revision. *In this thesis we would like to investigate the application of the TMS in find invalidated cached queries in agent programming platforms.*

Ontology debugging (axiom pinpointing) is the process of finding explanations for an error (a derivation) in an ontology. After making further review of existing approaches to ontology debugging/axiom pinpointing, it is even more interesting as many works on this topic such as in [57, 59, 64, 81] adopted a glass-box approach, which is essential keeping track of the reasoner inferences and collecting assumptions leading to contradictions. This is exactly what an Assumption-based TMS (ATMS) [23] can do in its original form. The difference is that instead of building a dependency graph, the approaches mentioned above use the tree-like structure of tableaux reasoning methods. *Therefore, it is an open research question whether a TMS such as the ATMS can be applied to solve the problem of ontology debugging/axiom pinpointing*. The ATMS in its original form only supports Horn-clause inferences, and hence will not allow disjunctions, which is essential to many useful Description Logics. Therefore, it is reasonable to start with a problem where only Horn-clauses are allowed, and then extend the ATMS's ability to deal with more expressive logics, e.g., one with disjunctions (and possibly loops) or one which is undecidable.

[2]JPL: `http://www.swi-prolog.org/packages/jpl/java_api`

1.3 Research Objectives

The objectives this thesis aims to address are as follows.

- To apply a light-weight Truth Maintenance System to keep track of the dependencies between queries and facts in the agent's knowledge base. These dependencies will be used to allow query caching in agent programming platforms.
- To investigate the application of an ATMS to find incorrect mappings between Geo-spatial Knowledge-based Systems.
- To extend the ATMS to pinpoint axioms for ontologies encoded in a decidable but more expressive logic, the $\mathcal{ALC}$ description logic. In particular, the ATMS is extended to deal with disjunctions and cyclic terminology.
- To employ the extended ATMS in axiom pinpointing problem of an upper ontology, e.g., SUMO, which is represented by an undecidable logic, SUO-KIF.

1.4 Structure of the Thesis

The rest of this thesis is divided into two main parts. The first part is the literature review, including Chapter 3, 2, and 4. The second part of the thesis, presenting the main contributions, includes Chapters 5, 6, 7, and 8 . A summary of the chapters in this thesis is as follows.

Chapter 2 (Intelligent Agents and Knowledge-based Systems) gives a brief overview of Intelligent Agents and Knowledge-based Systems. The Belief-Desire-Intention (BDI) model and some agent programming languages and platforms are also introduced.

Chapter 3 (Description Logics and Ontology Debugging) introduces Description Logics and previous approaches to the ontology debugging/axiom pinpointing problem. In this chapter, the $\mathcal{ALC}$ logic is used as an example of reasoning services in Description Logics, including the problem of ontology debugging and axiom pinpointing.

Chapter 4 (Truth Maintenance Systems) gives an overview of Truth Maintenance Systems and their application in Knowledge-Based Systems. Two popular types of TMSs are discussed, the JTMS and the ATMS. Their differences and applications are also presented.

Chapter 5 (Query Caching in Agent Programs) presents an approach to query caching in agent programs using a light-weight TMS. This chapter is partly based on the joint work in [1]. My contribution is the development of query caching in the GOAL programming language which uses SWI-Prolog as the Knowledge Representation Technology.

Chapter 6 (Detecting Geospatial Ontology Mapping Errors) shows how to use an ATMS to find errors in auto-generated ontology mappings. The geospatial data are stored in a knowledge-based system supporting the Logic of NEAR and FAR. In this chapter, we apply an ATMS to find all incorrect instance-mappings between two geospatial ontologies. This chapter is partly based on the work in [66].

Chapter 7 (Debugging Ontologies with Disjunctions and Loops) presents an extension of the ATMS to deal with non-Horn clauses, e.g., in ontologies with disjunctions and loops. The chapter also presents results of experiments comparing the performance of the extended ATMS and two Description Logics reasoners, Pellet and MUPSter. This chapter is mainly based on the paper [67].

Chapter 8 (Axiom Pinpointing for SUMO) applies the extended ATMS in the previous chapter to address the axiom pinpointing problem in SUMO, a widely-used upper ontology.

Chapter 9 (Conclusion and Future Work) completes the thesis with a summary of contributions and potential directions to future work.

Chapter 2

Intelligent Agents and Knowledge-based Systems

2.1 Introduction

In this chapter, we provide a wider context of the research presented in this thesis. In particular, we review intelligent agents and knowledge-based systems (KBS). Intelligent agents perceive the environment and perform actions to accomplish their tasks. One approach to agent-based software is the Belief-Desire-Intention (BDI) model where the notions of beliefs, desires (goals), intentions (sequences of actions) are abstracted and represented in agent programming languages and platforms. After introducing the BDI model in Section 2.2.2, we briefly mention the popular agent programming languages and platforms and also give an example of an agent program in GOAL [48] in Section 2.2.3. To be able to achieved the goals efficiently, intelligent agents need to represent the domain knowledge and the perceptions of the environment in some form as well as to reason about them. Therefore, in the last section, we give a short overview of Knowledge-Based Systems (KBS), i.e., systems which can represent knowledge and exploit them using some reasoning mechanisms.

2.2 Intelligent Agents

2.2.1 Agents and the Environments

One definition of intelligent agents which is widely cited among Artificial Intelligence community is from [79]: *"An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators"*. The popularity of this definition is explained by its generality, i.e., it captures not only software agents but also other types of agents, including human agents. Basically, an agent can be considered as a mapping from the states of the agent's environment to a course of actions. The agent's environment may be real (e.g., robotic agents, human agents, etc.) or virtual (e.g., software agents), partially or fully observable (e.g., in a poker game or a chess game), and can also contain other agents (e.g., in a team of agents).

The relationship between an agent's environment and its actions is bi-directional as shown in Figure 2.1. In one direction, the agent uses information from the sensors about the current state of its environment to decide which actions to perform. In the other direction, the actions performed by an agent can change the environment. For example, consider a system consisting of several stock trading agents. If one agent decides to sell or buy some stocks then the environments (e.g., stock prices and quantities) will change. This environment belongs to not only the agent performing the actions but also other agents in the system, and hence will affect future actions of all agents.

However, a question arises: how can the agent choose which actions to perform? So far we assume that the agent is only a straightforward mapping from

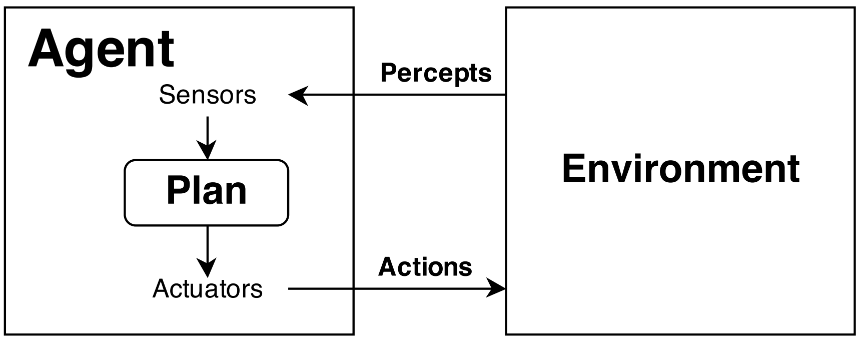

FIGURE 2.1: An agent and its environment. This figure is based on the one in [79].

an environment to some actions, i.e., it only reacts to the current environment. The agent can think about which actions to choose based on what it wants to achieve or to avoid. For example, a Google autonomous car agent may want to achieve a task of getting to a place and also avoid running out of fuel on the way. The process of perceiving the environment, thinking about which action to do, and performing an action in an agent forms a cycle, namely *sense-plan-act*, as illustrated in 2.1. This is often called a deliberation cycle [21].

2.2.2 BDI Model

The *sense-plan-act* cycle is often used in an agent software model called Belief-Desire-Intention (BDI), which abstracts human reasoning concepts. The model has been implemented in different agent and multi-agent systems platforms such as the Procedural Reasoning System (PRS) [40], 2APL [20], Jason [14], GOAL [47, 48], etc. Three main components of the BDI model are as follows.

Beliefs represent information about the environment from the agent's view.[1]
In logic-based BDI agents, beliefs are usually encoded as ground facts and domain rules. When an agent perceives its environment in a *sense-plan-act*

[1]Some might prefer to refer to knowledge rather than beliefs. However it would be less ambiguous to use the term *beliefs* rather than knowledge because a belief might be true or not at a time point.

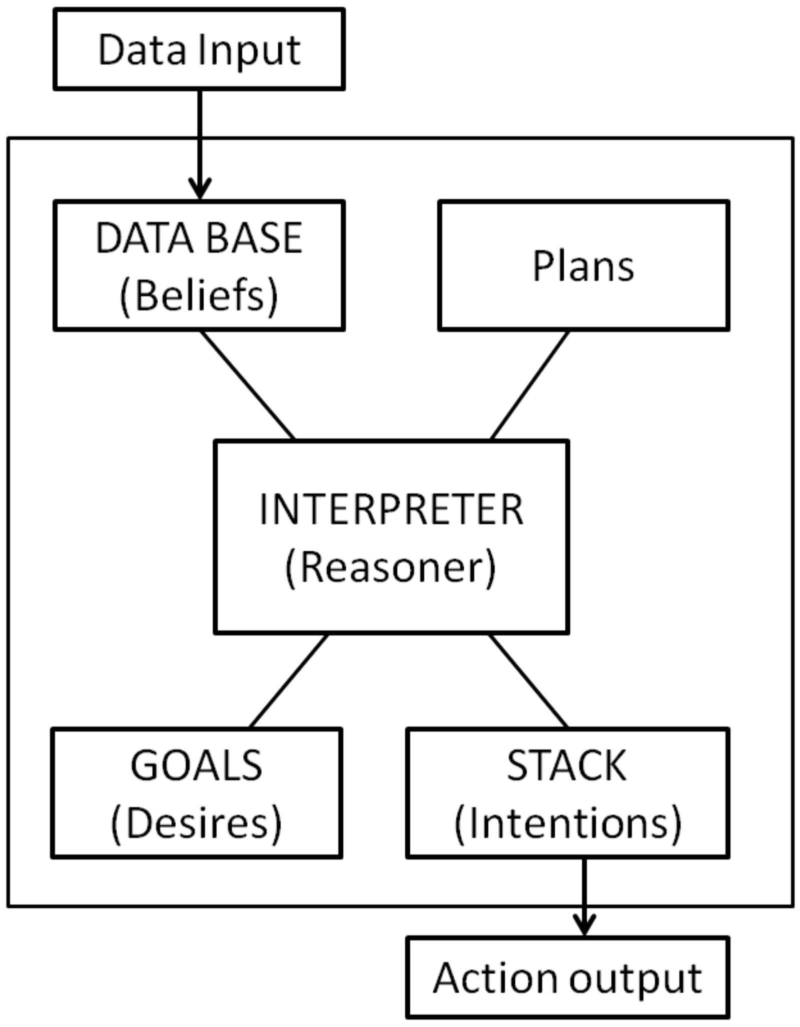

FIGURE 2.2: The Procedural Reasoning System (see [40]).

cycle, the beliefs are also updated accordingly. Note that not all beliefs can be perceived directly by sensors, instead some are implied by the agent from other beliefs. For example, a stock trading agent might acknowledge via sensors that the stock quantity is dropping for a particular stock and have a domain rule that if a stock quantity drops then its price will be higher. The agent will then infer that the stock price is higher and update its belief about stock prices.

Desires are essentially the goals which a BDI agent wants to achieve (a.k.a. achievement goals) or maintain (a.k.a. maintenance goals). Desires can be declarative such as the mental or physical states the agent wishes to reach.

In our example, the stock trading agent might want to own some amount of money, i.e., to reach to the state of owning that amount of money.

Intentions are selected courses of actions to achieve some desires given the agent's beliefs. In the PRS, intentions are active plans and stored in a process stack. Regarding the previous example of the stock trading agent, intentions can be sequences of buying and selling actions to which the agent commits to pursue its desires.

2.2.3 Agent Programming Languages

Even though agent and multi-agent systems, or more precisely the concepts and theories of agency, can be implemented in many programming languages, researchers have developed a range of agent programming languages and platforms which can help programmers to develop such systems more efficiently and easily. In [13], the authors provide a comprehensive survey of such programming languages and platforms. The programming languages for implementing agent and multi-agent systems can be either declarative, imperative, or a hybrid of these two approaches. Roughly speaking, a declarative agent programming language such as GOAL [48] specifies agent capabilities, beliefs, and goals without controlling explicitly what the agents should do to achieve their goals. In contrast, the imperative approach to implementing agent and multi-agent systems, e.g., the JACK Agent Language [91], uses or extends a traditional imperative language like Java or C with some features of logic languages to offer agent-specific abstractions. Some agent programming languages such as 2APL [20], Jason [14], etc., adopt a hybrid approach that can specify beliefs and goals in a declarative way while also employ the imperative approach on describing plans, control of flows, etc. For clarity and readability, in the rest of this thesis, we will provide examples and implementation in the GOAL agent programming language.

2.2.4 The GOAL Agent Programing Language

GOAL is basically a rule-based programming language written in Java. It is built on top of a Knowledge Representation Technology (KRT), e.g., SWI-Prolog [88]. The KRT provides GOAL with a mechanism to represent beliefs and goals as well as to reason about them using rules specified by GOAL programmers. Knowledge, beliefs, and goals in GOAL are declaratively defined using the KRT language such as Prolog, Datalog, etc. *Knowledge* is the set of rules and facts specifying the domain knowledge required by the agent to accomplish its tasks. These rules and facts are static, i.e., they do not change over time. *Beliefs*, on the other hand, are the set of facts representing the agent's perceptions of the environment. As the environment can change, so do the agent's beliefs. A feature distinguishing GOAL from other agent programming languages is *declarative goals*, i.e., GOAL programmers can declare *goals* as the states which the agent aims to reach. In GOAL, an achievement goal, `a-goal` implies that it is a goal the agent want to be, but the agent currently does not believe that this goal has been achieved. The rules in GOAL are grouped into modules. The *main module* provides the strategies to select an action. The action specification in the *init module* specifies the preconditions and post conditions of actions. The *event module* processes new percepts on the environment received by the agent and possibly updates the agent's beliefs accordingly.

```
init module {
   knowledge {
     block(X) :- on(X, _).
     clear(X) :- block(X), not( on(_, X) ).
     clear(table).
     tower([X]) :- on(X, table).
     tower([X, Y| T]) :- on(X, Y), tower([Y| T]).
   }
   beliefs    { on(b1,b2). on(b2,b3). on(b3,table) }
   goals    { on(b1,table), on(b2,table), on(b3,table)}
   actionspec {
     move(X, Y)
         { pre { clear(X), clear(Y), on(X, Z), not( on(X, Y) ) }
           post { not( on(X, Z) ), on(X, Y) } }
     }
}

main module [exit=nogoals]
{
 program{
 #define misplaced(X) a-goal(tower([X| T])).
 #define constructiveMove(X,Y) a-goal(tower([X,Y|T])),bel(tower([Y|T])).
      if constructiveMove(X, Y) then move(X, Y).
      if misplaced(X) then move(X, table).
 }
}
event module{
   program {
      forall bel(percept( on(X,Y)), not(on(X,Y))) do insert(on(X, Y)).
      forall bel(on(X,Y), not( percept(on(X,Y)))) do delete(on(X, Y)).
    }
 }
```

LISTING 2.1: Agent Program for Solving Blocks World Problems written in GOAL

An example of GOAL agent programs is given in Listing 2.1. This listing presents a GOAL agent program for solving Blocks World Problems. In Blocks World, the agent is given a set of blocks placed at some initial positions and a goal stating the positions of the blocks which the agent needs to achieve. The task is to move the blocks from the initial configuration (i.e., the initial beliefs) to the final configuration (i.e., the goal). As the agent is able to perceive changes in the environment using the event module, it can update its beliefs accordingly. In this example, achievement goals `a-goal` have been used to define a misplaced block and a constructive move. For example, a move is constructive if it helps the agent to get closer to the goal given its current beliefs.

A more thorough explanation for each part in a GOAL agent program is given in Section 5.2. Within the scope of this chapter, we only give a brief overview of the main components in a GOAL agent program. For more details on programming intelligent agents in GOAL, we refer the reader to [48].

2.3 Knowledge-based Systems

Most intelligent agent systems use some form of knowledge to solve their tasks, and hence they can be considered as a special class of systems, namely *knowledge-based systems*. Knowledge-based systems (KBS), as defined in [16], are systems *"for which the intentional stance is grounded by design in symbolic representations"*. The symbolic representations of knowledge in such systems are referred to as their *knowledge bases*. There are two important features of a knowledge-based system: the ability to represent knowledge and the ability to reason about knowledge.

Knowledge representation ability is particularly important for systems which required knowledge reuse, i.e., systems where the tasks are not fixed. For example, a medical system which can hold knowledge of diseases can use this knowledge to give recommendations to doctors as a decision support system. The knowledge can later be reused to generate medical reports. Also, as the knowledge of diseases have been conceptualised and represented in some form, it can adapt to various concrete circumstances, e.g., to different patients. Knowledge representation in a KBS correspond to a component of KBS, namely the Knowledge Base (KB). In a KB, knowledge are normally conceptualised in symbolic representation using some formalisms. Some well-known formalisms representing knowledge are first-order logics, description logics, Horn clauses[2], etc. Depending on the level of expressiveness required and the computational tasks, different KBS can choose different formalisms to represent their knowledge.

A formalism for knowledge representation is essential because of the need to reason about knowledge. We call *knowledge reasoning* the process of inferring the implicit knowledge from explicit ones, and this process is normally performed by an "inference engine" of the KBS. For example, the KB of Blocks World agent represented in Listing 2.1 of the previous section, uses Horn clauses as the formalism. Given the knowledge of Blocks World and the current beliefs about the environment, a Blocks World agent can use the inference engine (e.g.,, SWI-Prolog) to reason and then decide which actions to perform to achieve its goal. There are two main modes of reasoning in an inference engine, backward reasoning and forward reasoning. *Backward reasoning* (a.k.a. backward chaining or goal-driven reasoning) starts from the goal and and tries to find the concrete data supporting the goal (if such data exist). A famous implementation of backward reasoning is the PROLOG programming language [87]. In contrast, *forward reasoning* (a.k.a. forward chaining or data-driven reasoning) starts from concrete data and tries to derive as many data as possible using the reasoning rules. This mode of reasoning is usually employed in expert systems[3] to enrich their KB by adding implicit knowledge obtained during the reasoning process.

[2]Horn clauses are usually referred to as if-then rules or ground facts.

[3]An expert system, in general, is a system designed to solve complex tasks by reasoning about some domain knowledge.

2.4 Conclusion

In this chapter, we introduced intelligent agents and knowledge-based systems. We described the concept of intelligent agents and how they interact with their environments. We provided a brief overview of BDI model of agent-based software as well as the main approaches to agent programming languages and tools. In particular, we focused on the GOAL agent programming language as this background will be necessary for the demonstration and the implementation of the work in Chapter 5. Finally, we reviewed knowledge-based systems, their features, and their components so that it will be easier for readers to follow the next chapter, which focuses on a more specific knowledge-based system, the Description logic-based KBS.

Chapter 3

Description Logics and Ontology Debugging

3.1 Introduction

Description Logics (DL) [5] are a family of logics which are well-known for representing conceptual structures, i.e., high-level and structural descriptions of abstract objects. It has received much attention from researchers in Artificial Intelligence since the vision of the Semantic Web was formed [12] because DL has been used as a logical foundation for the future web. Apart from that, Description Logics have also been employed in a variety of applications such as software engineering, medicine, domain modelling, information systems, digital business, etc.

A DL-based KB system is often referred to DL-based ontologies, or more generally, ontologies. For example, Figure 3.1 shows a simple ontology describing some animals. Concepts $Animals$, $Sheep$, Cow, and $MadCow$ are describing a class of animals, a class of sheep, a class of cows, and a class of mad cows respectively. Axiom $Sheep \sqsubseteq Animal$ defines that $Sheep$ is a sub-class of $Animal$, i.e., a sheep is an animal. A more complicated concept definition, the definition of $MadCow$, states that a mad cow is a cow which eats either a sheep or a cow.

$$Sheep \sqsubseteq Animal$$
$$Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal$$
$$MadCow \sqsubseteq Cow \sqcap \exists eats.(Sheep \sqcup Cow)$$

FIGURE 3.1: A simple ontology describing animal concepts.

Beside representation ability, DL-based Knowledge Base Systems provide reasoning services to extract useful but implicit information from explicit data. Some standard reasoning services with ontologies include concept satisfiability test (i.e., checking if a concept could have any instance), subsumption (i.e., checking whether one concept subsumes another), classification (i.e., providing a hierarchical structure of concepts in the KB), instance checking (i.e., checking if a particular individual belongs to a concept)... Moreover, there are non-standard reasoning services which help users to create and manage ontologies more easily and efficiently. Among these are *axiom pinpointing* and *ontology debugging*. Axiom pinpointing [81] is a process of getting sets of axioms responsible for a given consequence of the ontology. This service is similar to explanations in other KB systems, i.e., explaining why something is derivable from the KBs. Ontology debugging is a special case of axiom pinpointing as it involves detecting the sources of inconsistency (i.e., semantic defects) in ontology. However, the role of a debugging procedure is usually not only to pinpoint the sources of inconsistency, but also to provide potential repairs. Currently, axiom pinpointing and ontology debugging services have not been integrated in most existing DL reasoners.[1] This chapter briefly introduces some background

[1]By the time of writing this report, only Pellet reasoner supports explanation service for ontologies. The upcoming version of RacerPro reasoner (RacerPro 2.0) has also been announced to provide similar features.

on Description Logic and current approaches to axiom pinpointing and ontology debugging in the literature.

The rest of this chapter is divided into three parts. Section 3.2 provides the preliminaries of a Description Logic together with the reasoning services. Section 3.3 explains two non-standard reasoning services which parts of the thesis aims to provide: axiom pinpointing and ontology debugging. Finally, Section 3.4 is an overview of the main approaches to ontology debugging and axiom pinpointing.

3.2 Description Logics - An Overview

This section introduces the syntax and semantics of a widely known Description Logics $\mathcal{ALC}$ [83]. Also, we present two main components of a DL-based Knowledge Base, TBox and ABox (a.k.a. the terminological and assertional parts of the ontology respectively). Finally, the main reasoning services of a DL-based KB, including the problem of axiom pinpointing and ontology debugging, are reviewed.

3.2.1 $\mathcal{ALC}$ Syntax and Semantics

Each description logic has a specific syntax based on which concepts are described. Given a set of atomic concept names and a set of atomic role names, the concept constructors could be used to describe new concepts. One popular description logic variant is $\mathcal{ALC}$, whose syntax and semantics are described below, as in [8].

Definition 3.1 ($\mathcal{ALC}$ Syntax)**.** An $\mathcal{ALC}$ concept description C could be either

- An atomic concept $A \in N_C$,

- $\top$, $\bot$,
- $\neg D$ (the complement of a concept),
- $C_1 \sqcup C_2$ (disjunction of two concepts),
- $C_1 \sqcap C_2$ (conjunction of two concepts),
- $\exists r.D$ (existential restriction), or
- $\forall r.D$ (universal restriction)

where N_C and N_R are the sets of atomic concept names and atomic role names, respectively, $r \in N_R$, and D, C_1, C_2 are concept descriptions.

Definition 3.2 ($\mathcal{ALC}$ Semantics)**.** An interpretation is of the form $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ (the domain) is a non-empty set of individuals and the interpretation function $\cdot^{\mathcal{I}}$ will interpret each concept name C as a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and every role name r as a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. Given that $r \in N_R$ and D, C_1, C_2 are concept descriptions, the semantics of the concept constructors defined in Definition 3.1 are as follows.

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, $\bot = \emptyset$
- $(\neg D)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus D$ (the complement of a concept)
- $(C_1 \sqcup C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ (disjunction of two concepts)
- $(C_1 \sqcap C_2)^{\mathcal{I}} = C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ (conjunction of two concepts)
- $(\exists r.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y : (x,y) \in r^{\mathcal{I}} \wedge y \in D^{\mathcal{I}}\}$ (existential restriction)
- $(\forall r.D)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y : (x,y) \in r^{\mathcal{I}} \rightarrow y \in D^{\mathcal{I}}\}$ (universal restriction)

Basically, the *top* ($\top$) and *bottom* ($\bot$) represents the domain and the empty set. The *complement* of a concept (e.g., $\neg D$) is interpreted as everything in the domain which is not in the set. Concept *conjunctions* (or *disjunctions*) are to describe the sets of individuals that belong to both concepts (or either of them). *Existential restrictions* are used to express concepts which have at least one $r-$role link to a specific concept. On the other side, *universal restrictions* (a.k.a. value restrictions) are constraints stating that if a concept has $r-$role links, all of them must be linked to a specific concept. For example, the concept description

$$Person \sqcap \exists lives.BritishCity$$

describes a person who lives in a British city (but they can live in other non-British cities as well). Similarly, a person who only robs the rich or never attacks good people can also be described as

$$Person \sqcap (\forall robs.RichPerson \sqcup \neg\exists attacks.GoodPerson).$$

Note that a concept description can be written in *Negation Normal Form* (NNF) where the negation (i.e., $\neg$) is moved to the most inner concepts (e.g., the previous concept description could be rewritten as $Person \sqcap (\forall robs.RichPerson \sqcup \forall attacks.\neg GoodPerson)$).

$\mathcal{ALC}$ is a basic description logic which could be extended to more expressive DLs by adding more concept and role constructors (e.g., number restrictions, role restrictions, etc). For instance, with number restrictions, the language can describe the concept of a binary tree as $\leq_2 hasChild.BinaryTree$.

Depending on the type of applications, one can choose a DL with a trade-off between tractability and expressiveness. With number restrictions, transitive roles and inverse role restrictions, $\mathcal{ALC}$ can be extended to $\mathcal{SHIQ}$, the logic behind the simplest variant of the Web Ontology Language (i.e., OWL Lite). This description logic is supported in most DL reasoners such as Pellet [86], Fact [49], and RacerPro [45]. For ontologies which prefer the performance of reasoning services to expressiveness, there exist light-weight DLs such as $\mathcal{EL}$ [90], which allows only conjunctions and existential restrictions but provides polynomial time for standard reasoning tasks.

3.2.2 TBox and ABox

Normally, a DL-based ontology is divided into two parts: a terminological part which has concept definitions and an assertional part which contains facts about individuals, namely TBox and ABox. In this section, we provide descriptions of these components in a DL-based Knowledge Base and a short introduction to typical reasoning services with an ontology.

So far we can express concept descriptions using constructors mentioned above. However, as one might have very complex concept descriptions, it is really important to have *concept definitions* (i.e., to give names for concept expressions). For example, one could define a British City as a city which is a part of Britain by writing $BritishCity \equiv City \sqcap \exists isPartOf.Britain.$

Definition 3.3 (Concept definition)**.** A concept definition is a statement of the form $C \equiv D$ where C is a concept name and D is a concept description.

More generally, one can use the *general concept inclusion* (a.k.a. general inclusion) axiom to express the 'subclass-superclass' relationship between concept descriptions such as a city is a kind of populated place, or anyone who has a son is a parent. These statements can be encoded in $\mathcal{ALC}$ as follows:

$$City \sqsubseteq PopulatedPlace; Person \sqcap \exists hasSon.Person \sqsubseteq Parent.$$

Definition 3.4 (General concept inclusion)**.** A general concept inclusion (GCI) is a statement of the form $C \sqsubseteq D$ where C and D are concept descriptions.

It is clear that GCIs can be used to express concept definitions (e.g., $C \equiv D$ is the same as $C \sqsubseteq D$ and $D \sqsubseteq C$). The terminological part of a DL-based ontology consists of GCIs and is refered as the TBox.

Definition 3.5 (TBox)**.** A TBox is a finite set of GCIs. An interpretation $\mathcal{I}$ satisfies a TBox $\mathcal{T}$ if $\forall C \sqsubseteq D \in \mathcal{T}, C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Definition 3.6 (Unfoldable TBox)**.** For all $ax : C \sqsubseteq D \in \mathcal{T}$ where C is a concept name and D is a concept description, $\mathcal{T}$ is called *unfoldable* iff: C appears at most once in the left hand side of a concept definition (definitional); and D contains no direct or indirect reference to C(acyclic) .

One might think of TBox as the set of rules in a rule-based system. Then the question is where do the facts go? Basically, they are put in a part of the KB, namely the assertional part (or ABox for short). Some examples of axioms could be in an ABox are $BritishCity(Nottingham)$ (stating that Nottingham is a British city), $attacks(RobinHood, Sheriff)$ (presenting the fact that Robin Hood attacks the Sheriff), etc.

Definition 3.7 (ABox)**.** Let $a, b \in \Delta^{\mathcal{I}}$ (a, b are called individuals), r is a role description, and C is a concept description, an ABox $\mathcal{A}$ is a finite set of assertional axioms of the form $C(a)$ or $r(a, b)$.

An interpretation $\mathcal{I}$ satisfies $\mathcal{A}$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ holds for all $C(a) \in \mathcal{A}$ and $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ holds for all $r(a, b) \in \mathcal{A}$ where $.^{\mathcal{I}}$ maps every $a^{\mathcal{I}}$ to $a \in \Delta^{\mathcal{I}}$.

Definition 3.8 (DL-based knowledge base)**.** A DL-based knowledge base is a pair $KB = (\mathcal{T}, \mathcal{A})$ where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox. An interpretation $\mathcal{I}$ satisfies KB if its satisfies $\mathcal{T}$ and $\mathcal{A}$

3.2.3 Reasoning Tasks for a DL-based KB

Up to this point, one might wonder why there is such a distinction between TBox and ABox. The answer to that question lies in the different reasoning problems for TBox and ABox, so that treating them separately will make things clearer.

For the TBox, the typical reasoning tasks are subsumption checking and satisfiability testing. In fact, a subsumption problem can be reduced to a satisfiability test. For example, $C \sqsubseteq_{\mathcal{T}} D$ is equivalent to $C \sqcap \neg D$ is unsatisfiable w.r.t. $\mathcal{T}$. Therefore, it is enough to have a satisfiability test algorithm to perform reasoning tasks for the TBox.

Definition 3.9 (Concept subsumption)**.** Let $\mathcal{T}$ be a TBox and C, D concept descriptions, $C \sqsubseteq_{\mathcal{T}} D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$.

Definition 3.10 (Concept satisfiability)**.** Let $\mathcal{T}$ be a TBox and C a concept description, C is satisfiable w.r.t. $\mathcal{T}$ iff there exists an interpretation $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$.

Inference problems for ABox are consistency checking (Definition 3.11) and instance checking. Consistency testing shows whether there is any contradiction in a KB while instance checking tests if an individual a belongs to a concept C. One might observe the fact that the satisfiability problem could indeed be reduced to a consistency problem, i.e., C is satisfiable w.r.t. $\mathcal{T}$ iff $\mathcal{A} = \{C(a)\}$ is consistent w.r.t. $\mathcal{T}$. This observation is important for the ontology debugging tasks which we will cover in the following sections because roughly speaking, debugging a TBox with unsatisfiable concepts could be reduced to debugging an inconsistent ABox.

Definition 3.11 (ABox consistency)**.** An ABox $\mathcal{A}$ is consistent w.r.t. a TBox $\mathcal{T}$ iff it has a model that is also a model of $\mathcal{T}$.

Definition 3.12 (Instance checking)**.** An individual a is an instance of a concept description C w.r.t. $\mathcal{T}$ and $\mathcal{A}$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ for all models $\mathcal{I}$ of $\mathcal{T}$ and $\mathcal{A}$.

3.2.4 Tableau-based Reasoning

To solve reasoning problems in Description Logics, a method called tableau-based reasoning is frequently used due to the tree-model property of such languages, i.e., a model can be constructed as a tree-like structure. This section briefly describes a tableau-based reasoning procedure for solving reasoning problem in Description Logics. A more comprehensive review of tableau-based methods are given in [5], including full examples and optimisation techniques.

In general, a tableau is a set of rules, namely completion rules. These rules are used to decompose a formula into sub-formulae. Intuitively the completion rules try to build a model for the theory if such model exists. To do so, a tableau-based method builds a completion tree consisting of nodes. A node is a leaf-node if either there is no rule applicable to the formulae in that node (i.e., the node is completed), or it has a clash (explained later in Definition 3.17), i.e., a contradiction. A leaf-node with a clash means that it is not possible to construct a model using the formulae in this node. A completed leaf-node in the tree represents a model of the theory, so if such nodes exist, there is at least a model of the theory, i.e., it is consistent.

Before applying the completion rules, all formulae in a theory are normalised into a normal form so that they can be processed by the rules. Usually the Negational Normal Form (NNF) is used as it is more suitable to introduce contradictory formulae (clashes), e.g., $C(i)$ and $\neg C(i)$. A completion rule can be either deterministic or non-deterministic. A rule is non-deterministic if it can introduce alternatives, i.e., choices. Otherwise, the rule is deterministic. For example, a rule decomposing a conjunction is deterministic while a rule decomposing a disjunction is non-deterministic as any of the disjuncts can be used to build a model. When a non-deterministic rule is applied, a new node (i.e., a new branch branch) of the completion-tree is created. The application of the completion rules stops if either a model of the theory is found (i.e., there is at least one completed leaf-node), or there is not possibly any model for the theory (i.e., all leaf-nodes have at least one clash).

The reasoning tasks mentioned in Section 3.2.3 can be solved using a tableaux algorithm. For example, to solve the satisfiability testing of a concept description C w.r.t. $\mathcal{T}$, one can assume that there exists an individual i of C, and apply the completion rules with $C(i)$ and the normalised formulae in $\mathcal{T}$ to build a model. If such a model in which $i^{\mathcal{I}} \in C^{\mathcal{I}}$ exists then C is satisfiable w.r.t. $\mathcal{T}$.

3.3 Axiom Pinpointing and Debugging in a DL-based Ontology

3.3.1 Axiom Pinpointing

Useful ontologies are usually large. For instance, a well-known medical ontology SNOMED CT[2] has more than 370,000 subsumption axioms. Due to the large size of the ontologies, it is difficult for a human to understand the implicit relationships or to measure the effects while modifying the ontologies. Moreover, most concepts in such ontologies are created by experts through collaboration, and hence it would be very tricky for a normal ontology user to explain a consequence, i.e., to find the sets of axioms supporting that consequence.

[2]Systematized Nomenclature of Medicine - Clinical Terms

Even if this is possible, it is sometimes necessary to find not one but all the explanations why a consequence holds. The tasks of finding these explanations in DL-based ontologies is referred as *axiom pinpointing*. Generally, axiom pinpointing has two main applications. On the one hand, it helps users/authors to understand the ontology better without a very expertised knowledge (i.e., explanation). On the other hand, it helps users to resolve inconsistency in an ontology and to change an ontology so that unwanted axioms will be avoided (i.e., debugging). These are two important tasks in ontology engineering.

Although all DL-reasoners support the standard reasoning tasks mentioned above such as satisfiability test for TBoxes or consistency test for ABoxes, there still exists the need to produce explanations automatically as a non-standard reasoning task (the term non-standard reasoning service was firstly used for axiom pinpointing in [81]). Intuitively, an explanation for a consequence c in a DL-based KB $(\mathcal{T}, \mathcal{A})$ is a set of axioms $e \subseteq \mathcal{T}$ sufficient for deriving c. There can be exponentially many such explanations. It is only necessary to consider the *minimal* explanations, i.e., ones which do not subsume other explanations. These minimal explanations are also referred to *justifications* [55], *MUPS*[3] [81], *MinA*[4] [9], or *environments* [66]. For consistency, we will use the term "explanation" in the rest of this thesis.

Definition 3.13 (Logical entailment in a KB)**.** Given a $KB = (\mathcal{T}, \mathcal{A})$ and an axiom ax, $KB \models ax$ iff for every interpretation $\mathcal{I}$ satisfying $\mathcal{T}$, $\mathcal{I}$ satisfies ax.

Definition 3.14 (Explanation of a consequence)**.** Given a $KB = (\mathcal{T}, \mathcal{A})$ and an axiom c, the set of axioms $e \subseteq \mathcal{T}$ is an Explanation of c iff $(e, \mathcal{A}) \models c$ and for all $e' \subseteq e$, $(e', \mathcal{A}) \not\models c$.

It should be noticed that this definition of an explanation of a consequence is similar to the notion of justification for a sentence presented in [55]. The

[3]Minimal Unsatisfiable Preserving Sub-TBoxes
[4]Minimal Axiom Set

definition also corresponds to the Minimal Axiom Set of an input Γ w.r.t. a consequence-property P in [9]. For the case of MUPSs [81], because the author restricts the definition of explanations for only an unsatisfiable concept in a TBox, one could easily see that a MUPS for an unsatisfiable concept C is equivalent to an explanation e of the consequence $C \sqsubseteq \bot$ in our definition.

3.3.2 Ontology Debugging

Since the purpose of axiom pinpointing is to find explanations for an arbitrary consequence, one can use this service to pinpoint axioms responsible for a contradiction. This section introduces briefly the basics of ontology debugging, in particular the typical semantic defects occurring in an ontology.

Before going to the formal definitions of debugging an ontology, one might wonder how can an ontology have 'bugs'. There are three main reasons why ontologies have bugs and debugging them is not trivial. The first and most obvious source of bugs in an ontology is from modeller mistakes. The whole idea of ontologies is to give a common vocabulary, and hence it requires efforts from many modellers. The more modellers an ontology has, the more inconsistent it could become, not to mention that for open-sourced ontologies, not all modellers are experts. The second source of semantic defects in ontologies is migration from one ontology language to another. The third possible reason for inconsistency in an ontology is that it uses concepts from different upper ontologies, i.e., ontologies defining abstract concepts which will then be used by domain specific ontologies to define more concrete concepts for different applications. An example of merging two large upper ontologies such as SUMO[5] and

[5]SUMO (Suggested Upper Merged Ontology) is an ontology for abstract concepts, usually used with a domain ontology (see http://www.ontologyportal.org).

CYC[6] in the same document can lead to more than 1000 unsatisfiable concepts (see [80]).

Definition 3.15 (Incoherence)**.** A TBox $\mathcal{T}$ is **incoherent** iff there is at least one unsatisfiable concept in $\mathcal{T}$.

Definition 3.16 (Inconsistency)**.** A $KB = (\mathcal{T}, \mathcal{A})$ is **inconsistent** iff there is no model for it, i.e., there does not exist an interpretation $\mathcal{I}$ satisfying KB.

Recall in Section 3.2.3, it has been shown that the unsatisfiability test can be reduced to a consistency test. Therefore, the task of debugging an incoherent ontology can also be reduced to debugging an inconsistent KB, even though an ontology could be incoherent but consistent. For example, an ontology

$$KB_1 = (\{C \sqsubseteq D \sqcap \neg D\}, \{D(a)\})$$

is incoherent as C is unsatisfiable, but it is still consistent as there is a model for KB where $C^{\mathcal{I}} = \emptyset$.

Given an $\mathcal{ALC}$ ontology, the only form of contradiction is that an individual a belongs to a concept C and its complement $\neg C$ as there is no number restriction (otherwise, one has another kind of contradiction such as $C \sqsubseteq \leq_2 r.D \sqcap \geq_3 r.D$).

Definition 3.17 (Clash)**.** A pair of assertional axioms $(C(a), \neg C(a))$ is a clash in $KB = (\mathcal{T}, \mathcal{A})$ iff $KB \models C(a)$ and $KB \models \neg C(a)$.

From our point of view, the process of debugging ontology involves two parts. The first is to identify which sets of axioms are responsible for the inconsistency (i.e., pinpointing). The second step is to propose how can these axioms be modified to make the concept satisfiable, or to restore consistency to the Knowledge Base (KB) with respect to some particular criteria. Within the first step, one

[6]An upper ontology for general knowledge. The open-sourced version OpenCYC currently has about 50,000 concepts and millions of assertions (see http://opencyc.org).

could also prepare for the later repair step by annotating which parts of the problematic axioms really cause the contradictions as there can be very complicated concept descriptions containing concepts which are irrelevant to the inconsistency. For example, consider the definition of concept $MadCow$

$$MadCow \equiv Cow \sqcap \exists eats.((\exists partOf.Sheep) \sqcap Brain).$$

If $MadCow$ is an unsatisfiable concept, the sources of the unsatisfiability can come from different parts of the axioms (e.g., Cow, $Brain$, $\exists partOf.Sheep$, etc). If the source of errors is only from concept Cow, one does not need to remove other parts of the axioms because some implicit entailments can be lost. In other words, it is possible to generalise the problematic concept description to eliminate inconsistency in this case (e.g., making the axiom become $MadCow \equiv \exists eats.((\exists partOf.Sheep) \sqcap Brain).$). Also, in the case of multiple unsatisfiable concepts in an ontology, it is very useful to consider parts of an axiom instead of the whole axiom. Let us assume that concept Cow is the source of 10 contradictions and $Brain$ is the source of only one contradiction. Certainly, one would prefer removing Cow to removing $Brain$ from the axiom.

However, how to change axioms is still a controversial issue, as it depends on the type of application domain. In critical domains such as medicine, some concepts should not be automatically corrected because it is not really safe to automatically repair an ontology by modifying the concept descriptions without expertise. In a use case in Chapter 4, the modeller decided to remove all problematic axioms, although removing one axiom would have been enough to restore consistency. Therefore, minimal-change approaches to restore consistency do not always work. We believe that one can use the parts-of-axioms idea to give suggestions to users or to annotate the problematic axioms rather than allow an ontology to be repaired automatically.

3.4 Related Work on Ontology Debugging

In this section, there is a brief introduction to main approaches to ontology debugging and axiom pinpointing. Generally, work on ontology debugging and axiom pinpointing can be grouped into two main groups: *glass-box* and *black-box*.

Black-box approach uses the DL-reasoner as a black box to compute explanations.

Glass-box approach modifies internal structures of the DL-reasoner to annotate derivations to a given consequence.

3.4.1 Black-box Approach

Black-box methods do not need to be bound with any specific Description Logic, as they only use the reasoner as an external component. All DL-based reasoners can answer queries such as "Is a concept C satisfiable w.r.t. TBox $\mathcal{T}$?" (Satisfiability Test). If the answer is no, it means that some subsets $\mathcal{T}'$ of $\mathcal{T}$ can be the set of axioms responsible for the unsatisfiability of C. Otherwise, $\mathcal{T}$ is not the set of axioms responsible for C's unsatisfiability. However, assume that $Sat(\mathcal{T}, C)$ is the C satisfiability test function w.r.t. $\mathcal{T}$, the real matter is to find the minimal set of axioms responsible for the C-unsatisfiability if $Sat(\mathcal{T}, C) = false$. To do so, one can start with the empty TBox $\mathcal{T}' = \emptyset$ and insert axioms into $\mathcal{T}'$ as long as $Sat(\mathcal{T}', C) = true$ (*expanding step*). At the point where $Sat(\mathcal{T}', C) = false$, the *shrinking step* can be performed. All axioms in $\mathcal{T}'$ are removed apart from the ones that removing them can make $Sat(\mathit{tbox}', C) = true$, i.e., ones are really essential for $Sat(\mathcal{T}, C) = true$. Note that by this strategy, one can only find one minimal set of axioms responsible for C-unsatisfiability.

Obviously, one might prefer not to apply the expanding step mentioned above. Instead they set $\mathcal{T}' = \mathcal{T}$ and then do the shrinking. However, this is very inefficient because the explanations for a contradiction are usually very small compared to the size of the original TBox. Therefore, the main concern while using black-box methods is to choose which $\mathcal{T}'$ to start shrinking at the beginning. If a reasonably small $\mathcal{T}'$ is used for shrinking, black-box approaches will be very advantageous because they do not restrict the expressiveness of the ontology languages, even though the completeness of the procedure cannot be guaranteed (i.e., cannot find all minimal explanations). Fortunately, by combining black-box calls with Reiter's Hitting Set Tree algorithm [75], it is possible to find all the explanations for concept unsatisfiability, according to the work reported in [56].[7]

For example, given a TBox $\mathcal{T} = \{ax_1, ax_2, ax_3, ax_4, ax_5\}$ where $Sat(\mathcal{T}, C) = false$ and there are two minimal explanations for the unsatisfiability of a concept C w.r.t. $\mathcal{T}$: $e_1 = \{ax_1, ax_2, ax_3\}; e_2 = \{ax_3, ax_4\}$. One can perform the shrinking step by removing each axiom from $\mathcal{T}$ in turn until $Sat(\mathcal{T}, C) = true$. Assume that at the beginning $\mathcal{T}' = \mathcal{T}$ and the axioms are removed in their ascending index order, the one will have the following transformations of $\mathcal{T}'$:

$\{ax_1, ax_2, ax_3, ax_4, ax_5\} \xrightarrow{1:\ \cancel{ax_1}} \{ax_2, ax_3, ax_4, ax_5\} \xrightarrow{2:\ \cancel{ax_2}} \{ax_3, ax_4, ax_5\} \xrightarrow{3} \{ax_3, ax_4, ax_5\} \xrightarrow{4}$
$\{ax_3, ax_4, ax_5\} \xrightarrow{5:\ \cancel{ax_5}} \{ax_3, ax_4\}$.

Note that only e_2 is found as the minimal set of axioms and if the axioms are removed in their descending index order then only e_1 is found. Also, in step 3 and step 4, ax_3 and ax_4 are not removed because removing them will make $Sat(\mathcal{T}', C) = true$. The example above uses a naive strategy, as it needs to loop through the whole TBox to find only one single minimal explanation. However, with a simple trick, one can also find all the minimal explanations. Assume that

[7]A hitting set of a collection C of conflict sets is a set H such that $\{H \cap c \neq \emptyset \mid c \in C\}$. A hitting set of C is minimal iff none of its subset is a hitting set of C.

after obtaining an explanation e, one firstly needs to consider the test $Sat(C, \mathcal{T}_i)$ where $\mathcal{T}_i \in \{\mathcal{T} \setminus \{ax_i\} \mid ax_i \in e\}$. $Sat(C, \mathcal{T}_i) = false$ implies that $\mathcal{T}_i$ still has some other explanations for the unsatisfiability of C w.r.t. $\mathcal{T}$.[8] The second step is to find the explanation for $Sat(C, \mathcal{T}_i) = false$. These steps are repeated until $Sat(C, \mathcal{T}_i) = true$ for all $\mathcal{T}_i$. In general, given EXPLAIN($C, \mathcal{T}$) as the function to find a single explanation for the unsatisfiability of C w.r.t. $\mathcal{T}$, the algorithm to find all explanations for the unsatisfiability of C w.r.t. $\mathcal{T}$ is as follows.

Algorithm 3.1 Find all explanations for a concept unsatisfiability given a function to find a single explanation

```
procedure EXPLAIN-ALL($C, \mathcal{T}, AllExplanations$)
    $e \leftarrow$ EXPLAIN($C, \mathcal{T}$)
    $AllExplanations \leftarrow AllExplanations \cup \{e\}$
    Let $T = \{\mathcal{T} \setminus \{ax_i\} \mid ax_i \in e,\ Sat(C, \mathcal{T} \setminus \{ax_i\}) = false\}$
    if $T = \emptyset$ then
        return $AllExplanations$
    end if
    for $\mathcal{T}_i \in T$ do EXPLAIN-ALL($C, \mathcal{T}_i, AllExplanations$)
    end for
end procedure
```

Note that in EXPLAIN-ALL, $AllExplanation$ is initially an empty set $\emptyset$ and accumulates gradually each time EXPLAIN is called. The algorithm presented above uses a similar idea as one in [56] to find all justifications for an OWL entailment. However, the authors in [56] use $Explain(C, \mathcal{T})$ to compute a Reiter's Hitting Set Tree (HST) [75] to find all the minimal justifications for the entailment. This approach benefits from optimisation techniques for the HST algorithm, and hence can be more efficient than our simple approach. Nevertheless, they all come from the basic idea, EXPLAIN-ALL($C, \mathcal{T}'$)$\subseteq$ EXPLAIN-ALL($C, \mathcal{T}$) if $\mathcal{T}' \subseteq \mathcal{T}$.

[8]Note that if C is unsatisfiable w.r.t. $\mathcal{T}' \subseteq \mathcal{T}$, it is also unsatisfiable w.r.t. $\mathcal{T}$.

3.4.2 Glass-box Approach

The Glass-box approach takes advantage of knowing the internal structure of the reasoner (i.e., the logic and language it provides), so that during the reasoning steps, it annotates an assertion with the set of axioms used to derive it. Schlobach et al. [81] were one of the first to attempt ontology debugging using this approach. In [81], the authors extend the tableau-based algorithm for testing concept satisfiability to pinpoint sets of axioms responsible for a concept unsatisfiability (referred as $MUPSes$). Every assertional axiom added into a node is associated with the sets of axioms used to derive it. When a clash is found, a minimisation function for a pair $(A(a), \neg A(a))$ is used to compute a formula φ, which is a disjunction of conjunctions of propositions ax_i (each conjunction $C : ax_1 \wedge \ldots \wedge ax_n$ is considered as a set $S = \{ax_i \mid C \models ax_i\}$ of axioms responsible for the clash). The found sets of axioms might be not minimal, and hence it needs to be minimised by finding the prime implicants of the minimisation function φ, i.e., the minimal conjunctions implying φ.[9] Each prime implicant is now a $MUPS$. From that, one can obtain $MUPS(C, \mathcal{T})$, which is the set of all minimal sets of axioms responsible for C-unsatisfiability w.r.t. TBox $\mathcal{T}$. The results reported in [81] were extended in [80] by applying Reiter's model-based diagnosis to find a set of diagnoses for an incoherent TBox. Given a set of $MUPSs$ w.r.t. a concept C and a TBox $\mathcal{T}$ as conflict sets, the author uses the Reiter's Minimal Hitting Set Tree Algorithm [75] to produce potential diagnoses for the unsatisfiability of C w.r.t. $\mathcal{T}$. In general, although the work in [80, 81] presents a complete framework to ontology debugging, there are still some issues such as it is restricted to only unfoldable $\mathcal{ALC}$ TBox and after producing diagnoses, it will be up to the user to choose which diagnoses (i.e., potential defects) they want to fix. Moreover, even though removing completely all axioms

[9]For example, given a boolean function $f(a, b, c, d) = abc + bcd + a + cd$, some implicants of f can be abc, bcd, a, cd (the number of implicants is in exponential to the number of variables). However, the prime implicants of f are only a and cd.

involved in a diagnosis can resolve the inconsistency, some diagnosis might not reflect the real problematic axioms.

Another way to look at the ontology debugging problem is to find the maximally consistent subsets of the ontology, as reported in [64]. This approach produces the same results as the combination between [81] and [80] to compute a set of maximally consistent sub-ontologies. The expansion rules for annotating assertions are similar to ones in [81]; however, the index-set I is used to represent a set of axioms responsible for an assertion instead of the minimisation function φ as in [81]. The main contribution of [64] is that the maximal satisfiable sets of axioms w.r.t. a TBox and an unsatisfiable concept are found immediately after firing the expansion rules (i.e., the axioms involved in a clash will be excluded when the clash is found), and hence it does not need an extra step to compute diagnoses as in [80]. The approach in [64] is then extended in [59, 60] to remove not the whole axioms but only parts of them. Also a refined blocking technique is proposed in [59] to deal with cyclic axioms so that termination is still guaranteed. A further contribution of [59, 60] involves measuring the impact of a change in a TBox axiom and classifying changes into two groups, helpful and harmful changes. Harmful changes will not remove the clashes, but can possibly lead to other clashes while helpful changes can not only resolve the contradiction but also recover some lost entailments.

Most of the work for ontology debugging mentioned so far is for the well-known $\mathcal{ALC}$ description logic described earlier in this chapter. To incorporate debugging tasks for ontologies in the Semantic Web, efforts have been made to provide debugging services for more expressive DLs such as ones underlying OWL[10]. Some work on explanation and debugging for OWL ontologies have been reported in [56, 57]. Using similar tableau-tracing methods as ones in [59, 64, 81], the authors define additional expansion rules to cope with

[10]Web Ontology Language

more expressive DLs such as rules for cardinality (i.e., $\leq$ and $\geq$ rules) and role-restrictions. Moreover, cycle blocking techniques have also been used to maintain the termination of the algorithm. More recently, Baader and Peñaloza [9] have proposed a generic tableau rule specification format and a pinpointing algorithm that works for reasoners specified in this format. They also show that termination of a tableau reasoner for satisfiability does not necessarily lead to the termination of its pinpointing extension. In addition, for tableau reasoners that require a blocking condition for termination, e.g., full $\mathcal{ALC}$, it is not sufficient for the pinpointing extension to use the same blocking condition as the reasoner, because the pinpointing extension needs to take into account not only the presence of an assertion in $\mathcal{A}$, but also its justifications to determine if a tableau rule instance should be blocked. In [9] they give a characterisation of a class of terminating tableaux where the blocking condition yields a complete and terminating pinpointing extension. However, to the best of our knowledge, this approach has not been implemented.

3.5 Conclusion

In this chapter, a short overview of Description Logics was given and the $\mathcal{ALC}$ logic has been used as an example. Besides, standard and non-standard reasoning services, including axiom pinpointing and ontology debugging, were introduced. The chapter ended with a literature review of previous work on DL-based ontology debugging/axiom pinpointing services, which are categorised into two main approaches: glass-box and black-box. The next chapter's topics include some background on Truth Maintenance Systems and how this technique is related to the problem of ontology debugging and axiom pinpointing.

Chapter 4

Truth Maintenance Systems

This chapter presents a popular technique in Knowledge-based Systems to keep track of inferences (dependencies) between data provided by a reasoner, namely the Truth Maintenance System. Firstly, some background on TMSs is given, including basic data structures in TMS implementations. The second part is an introduction to two popular types of TMS, namely Justification-based TMS and Assumption-based TMS, and show how they differ from each other. Finally, a brief overview of how Truth Maintenance Systems have been used in the area of Knowledge-based Systems is given.

4.1 Introduction

Truth Maintenance Systems (TMS), e.g., [30], also known as a Reason Maintenance Systems, are an approach to representing data and their dependencies derived by a reasoner (e.g., an inference engine or a problem solver). A TMS caches all inferences produced by the reasoner and represents them in its own data structures in forms of nodes and justifications. Using this representation of inferences and a set of operations, a TMS can perform tasks such as validating assumptions, maintaining consistency, or controlling reasoner searches, while at the same time keeping the reasoner focused on reasoning in task domains. A TMS is used together with a reasoner in a problem solver to find solutions given a set of assumptions, as in Fig. 4.1.

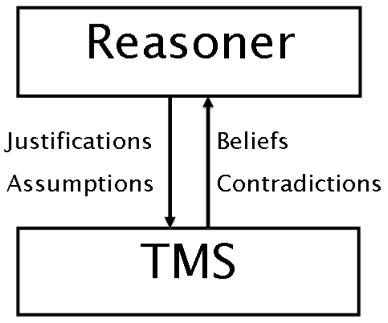

FIGURE 4.1: Communication between a TMS and the reasoner in a problem solver.

In general, one can look at a TMS as a directed graph, where nodes are either data or justifications for datum-nodes. In other words, a justification is a record of an inference, linking a datum node to the set of datum nodes deriving it. Using these recorded dependencies, a TMS allows a reasoner to quickly determine which nodes are "responsible" for belief in a particular datum.

According to [85], a TMS performs three main tasks:

1) given a derived datum, find the data or assumptions used to derive it;

2) given a set of assumptions, find all data can be derived from them; and

3) delete a datum and all the consequences which have been derived from it.

These tasks are also relevant to the problem of ontology debugging. For example, tracing the sources S_1 and S_2 of the assertions $A(x)$ and $\neg A(x)$, where A is a concept name and x is an individual in the ontology, gives the source of the contradiction (or clash) $S_1 \cup S_2$. Similarly, if one can find a minimal set of assumptions from which the contradictory assertions were derived, the minimal set of axioms which are the cause for the clash can also be identified.[1] This set corresponds to a *MUPS* in [81], or a *justification* for concept unsatisfiability defined in [55] (see Section 3.4.2).

4.2 Data Structures in a TMS

Different TMS implementations use different data structures to represent inferences and to perform their tasks, e.g., maintaining belief status, enabling/disabling assumptions, etc. However, most TMS implementations use the following data structures:

datum node a node in the dependency graph, supplied by the reasoner.

justification a justification connects datum nodes in the the graph, linking a set of supporting nodes (the antecedents) and a supported node (the consequence).

Definition 4.1 (Datum node)**.** A datum node n_{datum} is of the form

$$\langle datum, label, justifications \rangle,$$

where $datum$ is the formula given by the reasoner, $label$ represents status of the node (believed or unbelieved) or the set of nodes supporting the current node depending on the type of TMS, and $justifications$ store references to the justifications supporting this node.

Definition 4.2 (Justification for a node)**.** A justification for a node n_{datum} in a TMS is of the form

$$\langle n_{datum}, antecedents \rangle,$$

[1]In the literature on ontology debugging, the idea of tagging an assertion with the axioms used to derive it has also been proposed in [59, 64].

where $antecedents$ is a set of nodes supporting n_{datum}. A justification always has a consequence it justifies; however the set $antecedents$ may or may not be empty. If $antecedents$ of a justification is empty, this justification is supporting an foundational datum node (i.e., a node which is not derived from other nodes).

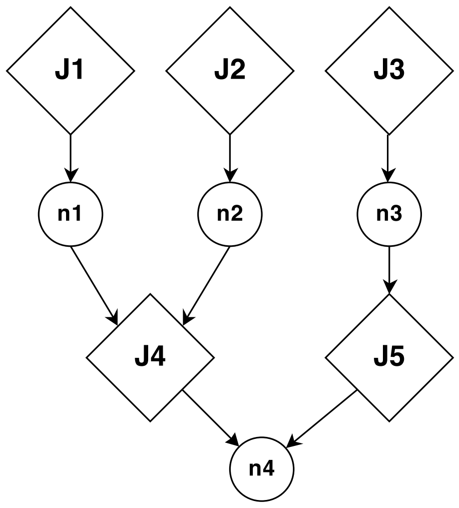

FIGURE 4.2: An example of the graph of datum nodes and justifications. Diamond and circles are justifications and nodes in the dependency network.

Figure 4.2 illustrates a dependency graph created by a TMS. n_i and J_i represent datum nodes and the justifications respectively. n_1, n_2, n_3 are foundational datum nodes because their justifications have an empty $antecedents$. A datum node can be justified by multiple justifications, e.g., n_4 is justified by both J_4 and J_5.

Based on datum nodes and justifications, a TMS supports the following basic operations [38]:

- create datum nodes and their justifications based on the data and inferences given by the reasoner;
- maintain the status of datum nodes by updating their $labels$; and
- when a contradiction is discovered, the TMS tells the reasoner about the contradiction. The TMS can also handle the contradiction by performing an operation, e.g., retracting an assumption leading to the contradiction.

Depending on the tasks and the type of coupling to the reasoner, there are different kinds of Truth Maintenance Systems. For instance, a Justification-based TMS (JTMS) can work in only one context[2] at a time while the Assumption-based TMS (ATMS) can work in multiple contexts. Similarly, while the JTMS and ATMS can support only definite clauses supplied by the reasoner, the Clause Management System (CMS) [76] can represent arbitrary propositional clauses. In the following sections, we will discuss two of the most popular types of Truth Maintenance Systems: Justification-based TMS (JTMS) and Assumption-based TMS (ATMS).

4.3 JTMS and ATMS: The Differences

The main differences between a JTMS and an ATMS are what it stores in a node's label and how the justifications for a datum node are maintained. Firstly, in the JTMS, since a node's label only stores the belief status, which is either *in* or *out*, one can only determine whether this datum is derivable or not from a particular set enabled assumptions A. Therefore, if a node is in given A, it holds in only one set of assumptions A (a.k.a. single context according to TMS literature). There is no direct way to check whether that node is still in when we change A without relabelling nodes' labels (by enabling and retracting assumptions). In contrast, the ATMS stores in each node's label the minimal sets of assumptions used to derive that node, i.e., multiple contexts. This approach will obviously cost more time and memory to maintain such sets of assumptions. However, in return the ATMS does not have to recompute the labels if the context changes. Whether a JTMS or an ATMS is a better choice depends on

[2]A context is a set of assumptions.

the type of application. If the application does not have many changes in contexts and only requires a single solution at a time, it would be better to choose the JTMS over the ATMS and vice versa.

Secondly, a JTMS only keeps a single valid justification as the supporting justification for a datum node at a time (explained further in Section 4.4.2). If the supporting justification of an in node becomes invalid, e.g., some of its antecedents are labelled *out*, the JTMS will try to find another supporting justification to keep its label as in. On the other hand, the ATMS maintains all justifications for a node at all time. This feature of the ATMS is particularly useful for applications which need to present all possible derivations of a datum at a time.

One interesting problem investigated in this thesis is finding sets of assumptions in which a datum holds or a contradiction occurs. We argue that it is more appropriate to use the ATMS for this task. Firstly, in the context of finding errors in a Knowledge-based System containing a set of assumptions A, because it is not certain which element of A is an error, one would need to try running the JTMS for each set $A' \subseteq A$ to see that whether node $n_\perp$ is in or out given the assumption set A'. The work for the reasoner and node relabelling can also be duplicated if the change in A' between two runs is small. This redundancy in relabelling in the JTMS can be avoided by using the ATMS. In this case, the ATMS can take the whole set of assumptions A and compute which subsets of A derive a datum. For example, it is possible for the ATMS to compute all minimal subsets of A which can cause a contradiction by examining the label of $n_\perp$ after termination. In addition, because of the four properties of node's label in the ATMS, one can determine immediately that a datum holds in an arbitrary set of assumptions A or not by checking the whether there exists an environment in that node's label subsumed by A without any relabelling effort. Moreover, because the ATMS stores all justifications for a node , it is much easier for the

ATMS to generate all explanations for a datum node or the contradiction node $n_\perp$ in forms of the justifications for that node.

4.4 Justification-based Truth Maintenance Systems

Although the TMS described in [30] is the first JTMS, [38] refer to it as the Non-monotonic JTMS due to its non-monotonic justifications. In the interests of simplicity, in what follows we describe the simplified version of JTMS presented in [38], i.e., non-monotonic justifications are not allowed, for simplicity and clarity. This JTMS only accepts propositional definite clauses as datums.

Definition 4.3 (Propositional Definite Clause)**.** A propositional definite clause is either

- an atomic clause (an atom) such as a; or
- a rule of the form $a \leftarrow b$ where a is an atom and b is either an atom or a conjunction of atoms.

4.4.1 Data Structures in a JTMS

A JTMS node is an *assumption node* if the reasoner explicitly tells the JTMS that the node is an assumption. It can be either enabled or retracted. An *enabled* assumption node is believed by the JTMS without the need for a valid justification. If an assumption node is *retracted* (i.e., not enabled), it is considered as a normal datum node, which will then only be believed if it has a satisfied justification. Initially, a JTMS contains a set of enabled assumption nodes A and a set of justifications J, which are given by the reasoner. Note that a JTMS never removes justifications and nodes, including non-enabled nodes, from the dependency graph.

The label of a node n stores the current status of n, which can be either in or out. If $A \cup J \vdash n$ under the propositional calculus rules then n is labelled in, i.e., node n is believed. Otherwise the node is labelled out (i.e., not believed).

4.4.2 Main Operations of a JTMS

Given the set of enabled assumption nodes A and the set of all justifications given from the reasoner J, the two main tasks of a JTMS are:

1. to return whether a particular node is labelled in with the current justifications and enabled assumptions.

2. to return an explanation for why a node is believed. This explanation is also called a *well-founded support* of the node and consists of all justifications used to derive it from the enabled assumptions. If there are multiple justifications for a node, the JTMS only chooses one valid justification[3] to be the *supporting justification* for that node. Note that the JTMS only returns **a single explanation**, which is the main difference to the ATMS which is introduced later in this chapter.

To perform these tasks, a typical JTMS supports three operations: adding a justification, enabling an assumption, and retracting (disabling) an assumption. *Adding a justification* requires the JTMS to check whether a node n which it justifies for is *in* or *out*. If it is *out* and all antecedents of the justifications is *in* then n is relabelled as *in* and the justification is marked as the new supporting justification for n. The procedure of relabelling nodes applies recursively to the nodes whose current supporting justification has n as an antecedent. The process terminates after all affected nodes are relabelled. An illustration of labelling an *out* node to *in* in a dependency network is given in Figure 4.3. Note that n_4 is

[3]A valid justification is the one whose all antecedents are labelled in.

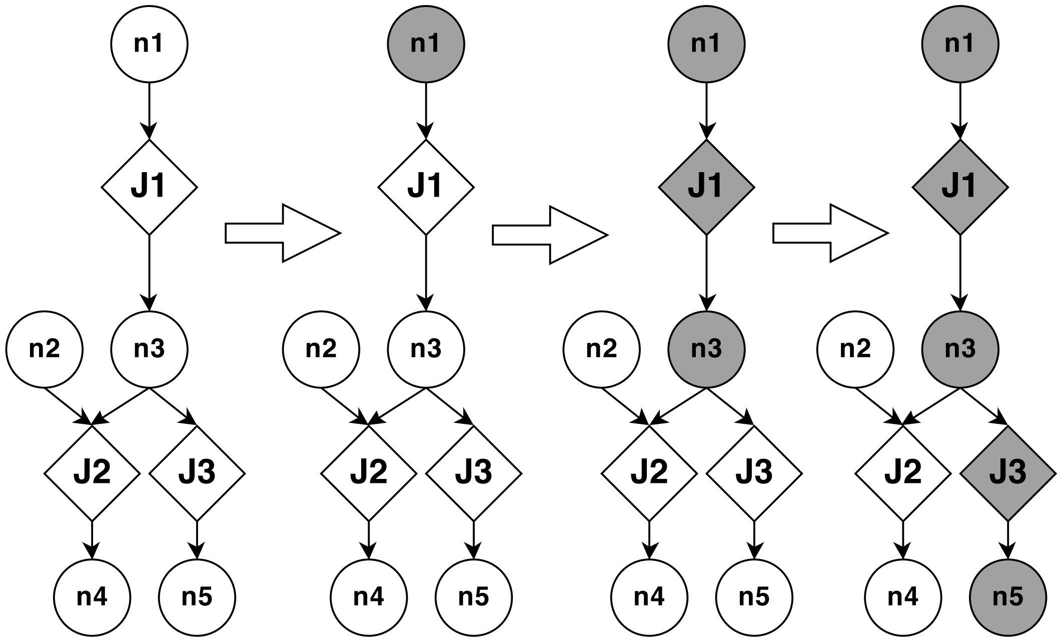

FIGURE 4.3: Labeling node $n1$ from *out* to *in*. White and dark-colour nodes represent *out* nodes and *in* respectively. Diamond and circles are justifications and nodes in the dependency network.

not relabelled because its supporting justification (J_2) is not satisfied (one of the antecedent, n_2, is still *out*).

Enabling and retracting an assumption are similar to adding a justification in the sense that they also, if necessary, involve relabelling nodes from *out* to *in* and *in* to *out* respectively. The main difference between enabling and retracting an assumption is that enabling assumptions only changes the labels of nodes from *out* to *in* and does not change a node's current supporting justifications, while retracting assumptions also needs to find another valid justification for a node (if such a justification exists) after labelling the node from *in* to *out*. In other words, as there is only one supporting justification for a node at a time, if the node is labelled *out* then the JTMS needs to find another valid justification to support it. If there is such an alternative justification supporting that node, the node will be re-labelled as *in*. Otherwise, its label is still *out*.

4.4.3 Representing Negation and Disjunction in a JTMS

As the JTMS only allows definite clauses, it is not possible to derive negation of a datum. To reason about negations, the JTMS has to employ some encoding tricks. Firstly, the JTMS represents a negation of a datum $datum$ as an independent node $n_{\neg datum}$ beside the node representing positive datum n_{datum}. Secondly, the JTMS add a justification of the form $n_{datum} \land n_{\neg datum} \Rightarrow \perp$. This justification means that if n_{datum} and $n_{\neg datum}$ are *in* together then there is a contradiction. If a contradiction occurs, the JTMS will signal the reasoner with the nodes leading to the contradiction. The reasoner has a contradiction-handler to process the contradictory data. This contradiction-handler can choose to just report the contradiction or perform a JTMS operation, e.g., retracting an assumption leading to the contradiction to restore consistency.

For disjunctive clauses such as $A \lor B$, the JTMS also needs to supply additional negation nodes $\neg A, \neg B$ and following justifications:

$$A \land \neg A \Rightarrow \perp,$$

$$B \land \neg B \Rightarrow \perp,$$

$$\neg A \land \neg B \Rightarrow \perp .$$

The first JTMS [30] also supports non-monotonic justifications of the form

$$\langle n_{datum}, inlist, outlist \rangle$$

in which n_{datum} is labelled in if all nodes in $inlist$ are *in* and all nodes in $outlist$ are *out*. In [30], an assumption node always has a non-monotonic justification with a non-empty $outlist$ supporting it. For instance, p is an assumption if n_p is justified by $\langle n_p, \{\}, \{n_{\neg p}\} \rangle$.

4.5 Assumption-based Truth Maintenance Systems

In this section, we present an overview of an ATMS including its data structures and algorithms. An Assumption-based Truth Maintenance System (ATMS) [23] also maintains a directed graph of datum nodes derived during the inference process. In general, given a set of assumptions A and a set of justifications produced by a reasoner, one can use an ATMS to determine all minimal subsets of A deriving a datum. An ATMS is also able to handle contradictions by marking contradictory sets of assumptions so that they cannot be used to derive any datum.

4.5.1 Structure of an ATMS node

An ATMS node n_{datum} is of the form:

$$\langle datum, label, justifications \rangle$$

where $datum$ is a propositional formula, $label$ is a set of *environments*, which are sets of assumptions used to derive that datum. Assumptions are explicit data from which implicit information can be inferred by the reasoner. The relationship between datum nodes in the dependency graph are represented by $justifications$ for datum nodes.

Each justification for a datum node n_{datum} is of the form:

$$\langle n_{datum}, antecedents \rangle$$

where $antecedents$ are datum nodes in the graph which immediately derive n_{datum}. Justifications are given to the ATMS by the reasoner. Since there are many ways a datum can be derived, it is possible to have multiple justifications for a particular node.[4]

There are four types of nodes in an ATMS, namely premise nodes, assumption nodes, datum nodes and contradiction nodes.[5]

- A node is a premise node if its label is of the form $\{\{\}\}$. Because premises hold in an empty environment, they hold universally. A justification for a premise node does not have any antecedent. Therefore, in the implementation, it is not necessary to maintain justifications for premises.
- An assumption node is a node justifying itself. For example, an assumption A may be represented as $\langle A, \{\{A\}\}, \{(A \Rightarrow A)\}\rangle$. As a result, an assumption node has at least one singleton environment (i.e., environments contains only one assumption) in its label. Note that it is also possible for an assumption node to have multiple justifications, i.e., to be derived from other nodes.
- A datum node, or a derived node, stores data derived during the inference process. A datum node n_{datum} with a non-empty label indicates that $datum$ holds in some environment. Some datum nodes are explicitly made to be assumptions by the reasoner in the beginning, which is similar to assumptions in the JTMS.
- The contradiction node $n_\perp$ represents falsity. A set of datums can derive a contradiction by deriving $\perp$, e.g., $n_p \wedge n_{\neg p} \Rightarrow n_\perp$. Then $n_\perp$'s label can be used to determine which environments (i.e., sets of assumptions) can lead to contradictions. In the ATMS, inconsistent environments (a.k.a. $nogoods$) are removed from all nodes labels except $n_\perp$.

[4]Note that the usage of environments and justifications in the ATMS are not the same. The former is to answer queries such as "given a set of assumptions A, will a datum hold in A?" while the latter is to maintain dependencies between ATMS nodes.

[5]In the original ATMS [23], the author also mentioned *assumed nodes*, which are not the assumptions by themself, but instead derived from assumption nodes. However, in this work, we consider them as normal datum nodes.

4.5.2 Properties of ATMS node labels

The main task of an ATMS is to ensure that each node's label (i.e., set of environments) is *minimal, consistent, sound,* and *complete*. In particular, these four properties of an ATMS node label are defined as follows:

minimality the datum of a node (so far) has not been discovered to be derivable from a strict subset of any set of assumptions in its label;

consistency if a set of assumptions is discovered to be inconsistent, then it is removed from the labels of all nodes (except $n_\perp$);

soundness if a set of assumptions is in the label of a node, then the reasoner has found a derivation of this node's datum which only uses those assumptions; and

completeness all ways of deriving the datum discovered by the reasoner so far are included in its node label.

Example 4.1. *Let us consider the following assumptions:*

1. $Swim$
2. $Rainy$
3. $HaveUmbrella$
4. $\neg HaveUmbrella$
5. $Rainy \wedge \neg HaveUmbrella \rightarrow GetWet$
6. $HaveUmbrella \rightarrow \neg GetWet.$
7. $Swim \rightarrow GetWet$

Using a forward chaining (data-driven) reasoner, it is possible to label datum $GetWet$ with the following environments (and possibly more):

- $e_1 : \{1, 7\}$,
- $e_2 : \{2, 4, 5\}$.

Firstly, note that there are exponentially many environments where a datum holds while it is only necessary to consider minimal environments of a datum. In Example 4.1, $GetWet$ also holds in environment $\{1, 2, 7\}$. However, because $e_1 \subset \{1, 2, 7\}$, one only needs to keep e_1 as an environment of $GetWet$. A label is minimal if it does not contain two environments, e and e', where $e \subset e'$.

Secondly, a node label needs to be consistent, i.e., no environment in its label is inconsistent. In an ATMS, there is a database called $nogood$[6] storing the set of unsubsumed inconsistent environments, e.g., the ones which can derive $contradictions$ like $\{3, 4\}$ in Example 4.1. A label is consistent if none of its environments subsumes an environment in $nogood$. For instance, a label containing all assumptions like $\{1, 2, 3, 4, 5, 6, 7\}$ is not consistent.

Thirdly, the label for a node n_{datum} must be sound, which means that for each environment e in n_{datum}'s label, $datum$ is derivable from e. In Example 4.1, the label of n_{GetWet} is sound because $GetWet$ is derivable from both e_1 and e_2.

Finally, every node's label has to be complete, i.e., for environment e where $datum$ is derivable, there has to be at least one environment e' in the label of n_{datum} such that $e' \subseteq e$. With the above example, the label of n_{GetWet} is complete because there does not exist an environment e where $GetWet$ is derivable and e is not a superset of either e_1 or e_2.

[6]In fact $nogood$ is the label of the contradiction node $n_{\perp}$ introduced in Section 4.5.1

4.5.3 Label Update Propagation Algorithms

The ATMS operates in a cycle as demonstrated in Figure 4.4. Initially, the reasoner sends information about assumptions to the ATMS, which then creates assumption nodes whose labels contain a single environment of one assumption (i.e., the assumption itself). As the reasoner informs the ATMS of new datum nodes and justifications, the ATMS *label propagation algorithms* update labels of previously asserted nodes to remove any subsumed environments (in the case of a justification for a datum node), or any environments which subsume an environment (in the case of a new justification for the distinguished node $n_\perp$ which represents contradiction). The process ends when the reasoner stops, i.e., no new justification has been created in the ATMS's dependency graph.

Clearly, the label propagation algorithms can be implemented in a naive way, in which for every new inference, new labels are completed created for each node. However this approach is not very efficient as it does not take into account the current labels and justifications of nodes in the dependency graph. In the next chapter, we will present an incremental approach based on the algorithms given in [25], in which only the latest label updates are computed and propagated to relevant nodes in the dependency graph.

4.5.4 Implementing Disjunctions in an ATMS

The ATMS as described in [23] does not support non-deterministic choices (i.e., disjunctions). However several approaches to handling disjunctions in an ATMS have been proposed in the literature. In [24] de Kleer extended the original ATMS to encode disjunctions of assumptions by introducing a set of hyper-resolution rules. However, such rules may significantly reduce the efficiency of the ATMS. Another approach [25] uses a justification for $\perp$ by negated assumptions to represent a disjunction of assumptions, e.g., $A \vee B$ can be encoded by

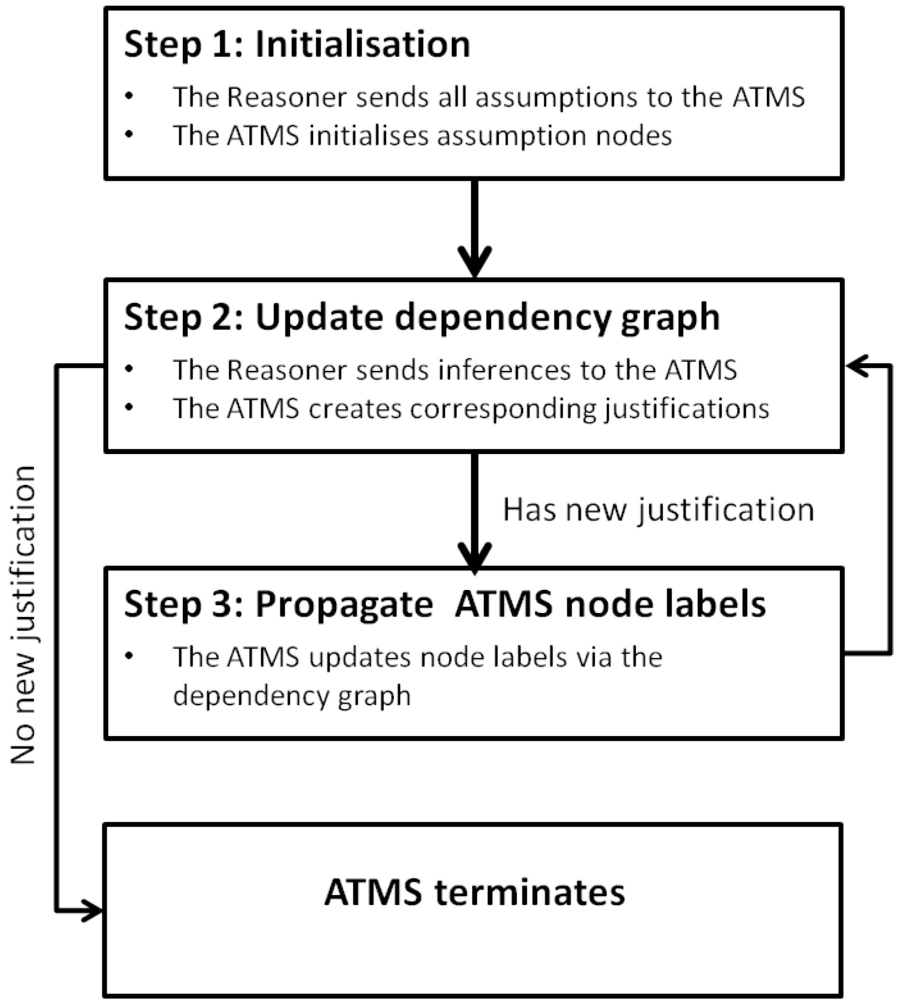
Step 1: Initialisation
The Reasoner sends all assumptions to the ATMS
The ATMS initialises assumption nodes
Step 2: Update dependency graph
The Reasoner sends inferences to the ATMS
The ATMS creates corresponding justifications
Has new justification
No new justification
Step 3: Propagate ATMS node labels
The ATMS updates node labels via the dependency graph
ATMS terminates

FIGURE 4.4: The ATMS operates in a cycle.

the justification $\neg A, \neg B \Rightarrow \perp$. Both of these approaches are limited to encoding a disjunction of assumptions.

In [76] the original ATMS was generalised to a clause management system (CMS) where justifications are arbitrary disjunctive clauses. To find the 'minimal support' for a clause, the CMS implementation described in [26] uses a method for computing prime implicants which relies on justifications being clauses consisting of literals to which the resolution rule can be applied.

4.6 Applications of Truth Maintenance Systems in Knowledge-Based Systems

Truth Maintenance Systems have been employed intensively in different domains, including Knowledge-Based (KB) systems. We take the definition of a KB system as given in [16]; that is a system *"whose ability derives in part from reasoning over explicitly represented knowledge"*. As one can see, there are two features of a KB system: knowledge representation and reasoning. From this point of view, a TMS can also be considered as a KB system in its own right. That is, a TMS has data structures to represent data and inferences and infers new knowledge via its assumption enabling/retracting operations (e.g., in the JTMS) or its label propagation (e.g., in the ATMS). However, in this section, we consider the TMS as only a component of a larger KB system. The main applications of TMS in KB systems are based on its ability to record data dependencies using the dependency graph and include belief revision, explanations/diagnoses generation, and incremental reasoning.

The main applications of the JTMS [30] include belief revision and non-monotonic reasoning. Belief revision is the process of changing a belief base (belief set) to adapt to new beliefs. The JTMS can achieve this task using its operations such as enabling and retracting an assumption. Truth Maintenance Systems create a style of belief revision, namely the *foundational approach*, which allow tractable revision and contraction of beliefs. Some examples of belief revision implementation following this approach are [3, 62]. TMS techniques are also implemented to maintain knowledge integrity of multi-agents sytems, i.e., each individual agent can have a local consistent knowledge base as in [52, 61] and data shared among agents can also be globally consistent [52]. The ATMS can also be used for belief revision as shown in [29], although it is not really necessary to implement belief revision using a multi-context system such as the ATMS.

The ATMS as introduced in [23], aimed to solve the problem of multiple (possibly contradictory) assumptions in qualitative reasoning which the JTMS was not capable of due to its single-context nature. Given the system components as assumptions together with the system descriptions and some measured observations, as in e.g., [75][7], the ATMS can diagnose faults in the system components in form of assumptions leading to contradictions [27]. This idea is extended to incorporate probable behaviour modes [28]. This approach has been applied in diagnosing many physical domains, including analog electronic circuits [19] and power transmission networks [31] . However, there has been relatively little work on diagnosing KB systems using the ATMS. One aim of this thesis is to investigate whether the ATMS can be used to solve the problem of fault-diagnosis in various KB systems.

With the introduction of the semantic web [12], Truth Maintenance Systems have also been used to revise semantic web systems. For example, there has been work on RDF-based systems such as revising consequences of an RDF[8] database after removing some statements [17]. In other work, the ATMS has been used to find minimal consistent subsets of OWL[9] documents $D_{sub} \subseteq D$ in a collection of OWL documents D which are sufficient to answer a query [43]. The justification structure in the JTMS is also exploited by [54] to generate explanations in for policy management in the AIR policy language (an RDF-based language). The explanations are presented in form of tree-like structures of justifications for a particular belief, which is produced by the JTMS's dependency tracking mechanism. More recently, there has been work on using TMS to optimise reasoning in ontology streams [77]. An ontology stream O^n_m from time-point m to timepoint n is a sequence of ontologies $O^n_m(m), \ldots, O^n_m(n)$ in

[7]In [75], a system is defined as a triple of <SYSTEM DESCRIPTION (SD), SYSTEM COMPONENTS (COMPS), OBSERVATIONS (OBS) >. The diagnosis task is to find a set $C \subset COMPS$ such that if C is removed from $COMPS$ then the system is no longer faulty.

[8]Resource Description Framework (see [73]).

[9]Web Ontology Language

which each ontology $O^n_m(i+1)$ ($m \leq i < n$ is a discreet time-point) is an immediate updated version of $O^n_m(i)$. The authors employ an approach similar to belief revision to cache ontology reasoning inferences in the original ontology of a stream in a JTMS so that later updates will only affect some parts of the original ontology (similar to adding justifications and retracting assumption in a JTMS), and new query answers can be computed more efficiently using the combination of cached inferences and recent updates.

4.7 Conclusion

In this chapter, we reviewed Truth Maintenance Systems and their applications. In particular, we examined two popular types of TMS, namely, the JTMS and the ATMS. For each type of TMS, the main data-structures and operations were given and the main differences between these two families of TMSs were summarised. We also discussed which type of TMS is more appropriate for particular tasks and gave a brief overview of previous applications of TMSs in Knowledge-based Systems found in the literature. In the following chapter, we will show how to employ the original ATMS to detect all errors in a Knowledge-based System.

Chapter 5

Query Caching in Agent Programs

5.1 Introduction

BDI[1]-based agent programming languages adopt the notions of beliefs, desires (a.k.a. goals), and actions to allow writing high-level, declarative agent programs. An agent programming platform therefore needs to represent and to reason about these notions in some knowledge representation technology (KRT). The interaction between an agent program and the KRT includes asking for answers to a query and updating explicit knowledge in its knowledge base. Query caching is a mechanism which allows agent programs to remember the results of previous queries so that the agent program does not have to resend such queries to the KRT. However, updates make changes to knowledge base, and hence may make previous cached results invalid. In this chapter, we develop a caching model which allows agent programs to cache query answers over multiple query-update cycles by using a light-weight truth maintenance system (TMS) to keep track of dependencies between queries and the facts used to derive the answers.

[1]Befief-Desire-Intention

The research questions, objectives, and the contributions of the work presented in this chapter are as follows.

Research Questions How can the data dependency graph maintained by a TMS can be employed to enable query caching in order to improve the performance of query answering in agent programming languages? How can such a TMS be constructed in a way that the overhead of maintaining (i.e., storing and invalidating) cached results does not outweigh the benefit of caching?

Research Objectives

1. To specify under which conditions query caching is beneficial to agent programs.
2. To implement a query caching facility for an agent programming language which adopts the TMS techniques (i.e., maintaining the dependency graph) so that only cached results which are affected by updates can be invalidated.
3. To verify the approach by evaluate the performance of query answering in different caching modes (i.e., without caching, with caching within a single query-update cycle, and with caching over multiple query-update cycles).

Contributions The main contribution is an implementation of query caching for GOAL agent programming language which allows users to choose various caching modes: without caching, with caching within a single query-update cycle, and with caching over multiple query-update cycles. The evaluation shows that caching query over multiple query-update cycles really improves query answering for agent programs significantly.

The rest of this chapter is organised as follows. In Section 5.2, we describe how an agent program interacts with its knowledge representation technology via query actions and update actions. We then look at how query caching can benefit agent programs in Section 5.3. This section presents two modes of query caching in agent programs, namely single-cycle and multi-cycle caching. In Section 5.4, we show how to implement the multi-cycle caching mode using a lightweight truth maintenance system. Section 5.5 presents experimental results in different query caching modes which show that query caching significantly improves the performance of agent programs.

5.2 Queries and Updates in Agent Reasoning Cycles

To be able to implement the notions of beliefs, desires, and intentions (recall Section 2.2.2), an agent programming platform should have a mechanism to represent and to use such notions. We refer to this mechanism as a Knowledge Representation Technology (KRT). A KRT can form part of the agent platform such as in the case of Jason and the PRS, or it can be an external component which interfaces with the core agent platform such as SWI-Prolog [88] in GOAL and JIProlog [53] in 2APL. Intuitively, a KRT can be considered as an inference engine (or a reasoner as in Figure 2.2).

The task of the KRT in an agent programming platform is to store the current state of the agent and its view of the environment in a database (i.e., the agent's belief base and goal base) and to infer implicit data given the current database. The agent program interacts with its KRT via two actions, *query* and *update*. For example, the stock trading agent might query against its belief base that which stock currently has the highest price and update the new price of a particular stock in its belief base. Note that query actions do not change the agent's databases while update actions do. The following sections demonstrate when and how query and update actions can occur. Note that depending on the agent programming language or platform, the agent program can send queries and updates to the KRT in different ways. For concreteness, in what follows we focus on the GOAL agent programming language, but similar operations are found in all logic-based BDI agent programming languages.

5.2.1 Queries

Recall from Section 2.2 that agents operate in a "sense-plan-act" cycle. In the *"plan"* phase of a logic-based BDI agent, the set of rules in the agent's program is executed. The antecedents of a rule are queries against the agent's beliefs and goals. An example of the rules in a Blocks World agent written in the GOAL agent programming language is given in Listing 5.1. In this example, the first rule contains two queries, one is against the goal base

```
a-goal(tower([X,Y|T]))
```

while the other is against the belief base (e.g.,

```
bel(tower([Y|T]))
```

).

```
main module{
  program{
    if a-goal(tower([X,Y|T])), bel(tower([Y|T])) then move(X,Y).
    if a-goal(tower([X|T])) then move(X,table).
  }
}
```

LISTING 5.1: Rules in Blocks World agent program written in GOAL language

Similarly, queries also appear in parts of the agent programs which use rules to represent domain knowledge such as ones in Listing 5.2. For example, the agent considers a block X clear if X is a block and there is nothing on top of X. If the agent program sends a query `clear(X)` to the KRT (e.g., SWI-Prolog for GOAL) then the KRT applies the domain rule

```
clear(X) :- block(X), not(on(Y,X)).
```

against the current belief base and check whether there is any answer (e.g., bindings of `X`).

```
clear(X) :- block(X), not(on(Y,X)).
tower([X]) :- on(X,table).
tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
```

LISTING 5.2: Rules in Blocks World agent program written in GOAL language

5.2.2 Updates

The second type of operation that the agent program uses to interact with its KRT is an *update action*. An update can happen in either the *"sense"* phase, i.e., when the agent perceives changes in the environment and update its beliefs or goals accordingly, or in the *"act"* phase, where the agent directly changes its databases via an internal action (e.g., an action which changes the agent's beliefs or goals but does not directly affect its environment). The following examples demonstrate these cases by database updates performed by a GOAL Blocks World (BW) Agent.

In a GOAL agent, all updates are instantiated (ground facts). In other words, an update of the form $p(\bar{X})$ where p is a predicate and $\bar{X}$ is a list of constants.

For instance, adding `on(X,Y)` into the belief base means adding `on(a,b)` where `a` and `b` are constants.

```
event module{
 program{
   forall bel( percept(on(X,Y)),on(X,Z),not(Y=Z))
        do insert(on(X,Y),not(on(X,Z))).
  }
}
```

LISTING 5.3: Updates in “sense” phase: event module of GOAL BW agent

As an example of the first case (i.e., updates in the “sense” phase), the *event module* in GOAL specifies how the agent program updates its belief base after being notified a change in the environment. Listing 5.3 shows a fragment of GOAL code which states that if the BW agent receives a percept that `on(X,Y)` and its current beliefs includes `on(X,Z)` and `Y` and `Z` are two different objects, then the agent should update its beliefs by adding a new fact `on(X,Y)` and removing the fact that `X` is on `Z`. Note that in this case, the update contains a sequence of updates, i.e., an addition and a deletion, while in other cases, there could be only one update.

The second case is where an update is performed directly by an action of the agent, and is demonstrated in the action specification of the BW agent as in Listing 5.4. The BW agent's “move” action has the *postcondition* which contains two updates to the agent's databases (the belief base and the goal base[2]): deleting `on(X,Z)` and adding `on(X,Y)`.[3]

[2]In GOAL, if a goal is achieved completed (all its sub-goals are also achieved) then it will be removed from goal base.

[3]Note that in GOAL, negative and positive literals in an update form a *delete* list and an *add* list respectively. An update to the database is performed by firstly delete all literals in the *delete* list from the belief base and then adding all literals of the *add* list into the belief base.

```
actionspec{
  move(X,Y) {
    pre{ clear(X), clear(Y), on(X,Z), not(X=Y) }
    post{ not(on(X,Z)), on(X,Y) }
}
}
```

LISTING 5.4: Updates in "act" phase: action specifications of BW agent (GOAL)

5.2.3 Agent Query-Update Cycles

From the previous sections, one can see that the interaction of an agent program with its KRT is basically a sequence of queries and updates. For example, an example log of queries and updates sent from the BW agent to its KRT (e.g., SWI-Prolog) is given in Listing 5.5.

```
...
query beliefbase:percept(block(X))
query beliefbase:not(block(X))
add beliefbase:block(b)
query beliefbase:on(f, table)
query beliefbase:on(e, f)
query beliefbase:on(e, f)
del beliefbase: on(a,b)
...
```

LISTING 5.5: An example log of queries and updates in the BW agent.

In Listing 5.5, the line

```
query beliefbase:on(f,table)
```

is a query to check if a fact `block(f,table)` exists in the belief base while the line

```
query beliefbase:percept(block(X))
```

is a query to SWI-Prolog which asks if there is any `X` such that `percept(block(X))` holds. The former query returns a boolean value while the later returns the list of substitutions (if any), i.e., mappings from `X` to a constant. The line

`add beliefbase:  block(b)`[4]

represents an addition of a fact. Note that in GOAL, an update requires an insertion of negative literals such as the one in Listing 5.3 will be transformed into the removal of the corresponding positive literals from the database under the Closed World Assumption (see [74]). For example, `insert(not(on(a,b)))` will be translated into `del(on(a,b))`.

Given all queries and updates sent to the KRT from the agent program, one can group the sequence of queries and updates into cycles. Such a cycle only contains consecutive queries and consecutive updates. Within a query-update cycle, a sequence of consecutive queries and a sequence of consecutive updates are denoted as the *query phase* and the *update phase* of the cycle respectively. For instance, Listing 5.5 can be considered as two query-update cycles as in Figure 5.1. The idea of dividing the queries/updates into phases comes from the fact that within a query phase, the agent's databases do not change. Therefore, it is only necessary to perform a query once, and keep that answer for later if the

[4]The real updates to SWI-Prolog are via `assert` and `retract` predicates for addition and deletion of facts respectively.

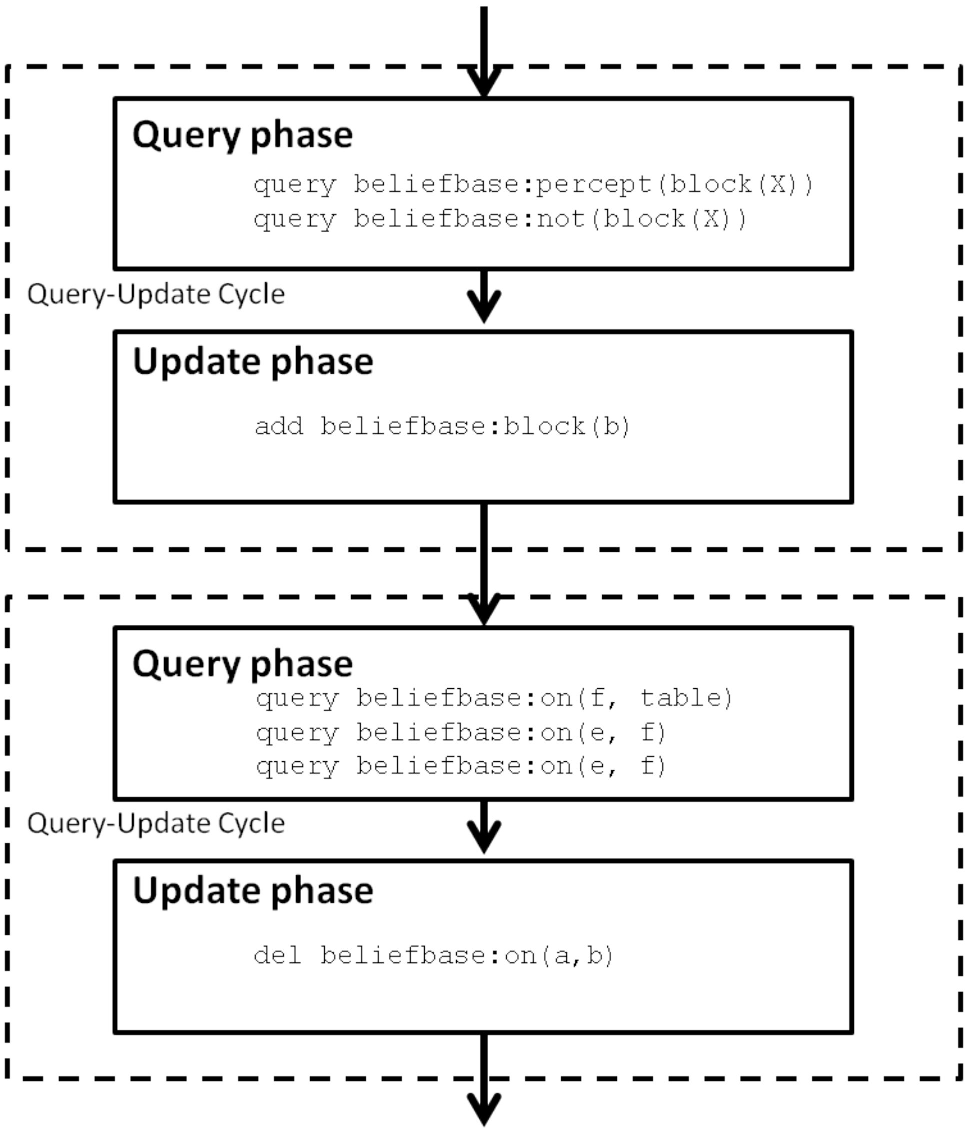
Query phase
query beliefbase:percept(block(X))
query beliefbase:not(block(X))
Query-Update Cycle
Update phase
add beliefbase:block(b)
Query phase
query beliefbase:on(f, table)
query beliefbase:on(e, f)
query beliefbase:on(e, f)
Query-Update Cycle
Update phase
del beliefbase:on(a,b)

FIGURE 5.1: An example of query-update cycle in the BW agent
.

same query is asked again within that query phase. This technique is called "caching".

By looking at the query/update pattern in each individual query-update cycle or in all cycles together in agent program executions, one can have an idea of whether caching query results can really improve the performance of agent programs. In [2], the authors have conducted such experiments with different combinations of agent platforms such as Jason, 2APL, and GOAL and task environments such as the Blocks World [79], Elevator Simulation [35], and the

Multi-Agent Programming Contests [11, 22]. The investigation of the agent query/update pattern gives following observations.

Observation 5.1. *In a single query-update cycle, the same query is performed more than once.*

Observation 5.1 comes from the fact that in all combinations of agent programs/platforms/environments, it is a consistent pattern that there are a number of queries repeated in a single query-update cycle. The ratio N/K where N is the total number of queries and K is the number of unique queries in a cycle ranges from 1.16 to 38.63, which means that the percentage of queries which are repeated within a cycle ($\frac{N-K}{N}$) ranges from 13.8% to 97.4%. Given this observation, it it clear that caching queries within a single query-update cycle can possibly improve the time which an agent program spends on querying the KRT.

Observation 5.2. *A significant number of queries are repeated at subsequent query-update cycles.*

Observation 5.3. *The number of updates U (add, deletes) performed in a query-update cycle is significantly smaller than the number of unique queries K performed in that cycle, i.e. $K \gg U$.*

Obsevation 5.2 is based on the average percentage of queries which are repeated in the query phase of next cycle. This number, called p in [2], ranges from 52% to 92%. This observation, together with Observation 5.3 which states that the number of queries is significantly greater than the number of updates with in a cycle, make it intuitive that it might be beneficial to cache queries over multiple query-update cycles as well. The reason why have Observation 5.3 is important for multi-cycle caching is that updates change the agent databases. Therefore, the fewer updates compared to queries within the same cycle, the fewer queries affected by these updates and the more useful query caching becomes.

5.3 Query Caching Modes: Single-Cycle vs. Multi-Cycle

In this section, we examine when query caching can improve the performance of agent programs and the algorithms for implementing two different query caching modes, single-cycle and multi-cycle. Finally, we show how a lightweight truth maintenance system can be used to implement multi-cycle query caching.

5.3.1 The Abstract Performance Model of Query Caching

The first question to ask before implementing query caching in an agent platform is when will caching be useful? Obviously, caching is useful only if the querying/updating time is smaller with caching than without caching. Here we will try to model the time spent in each agent query-update cycle without and with caching using the abstract performance model given in [2] with the following parameters.

- N: the average total number of queries per query-update cycle.
- U: the average total number of updates per query-update cycle.
- K: the average number of unique queries per query-update cycle.
- c_q: the average cost (time spent) per query.
- c_u: the average cost (time spent) per update.
- c_{ins}: the average cost (time spent) per cache insertion. Note that as a cache can be implemented using a hash table, c_{ins} is constant.
- c_{hit}: the average cost (time spent) per cache lookup. The word "hit" does not necessarily mean that it is a cache hit, i.e., C_{hit} also counts the lookup cost even when there is no hit in the cache.

- p: the percentage of queries repeated in the next (consecutive) query-update cycle. This is the ratio of the average number of queries in cycle i that reoccur in cycle $i+1$ to the average total number of queries per cycle (N).

Given the above parameters, the time cost per query-update cycle without any caching is:

$$cost_{no\_caching} = N \times c_q + U \times c_u. \tag{5.1}$$

Definition 5.1 (Cost Difference)**.** Let us denote $d_{M_1}^{M_2}$ as the cost-difference between a caching mode M_1 and a caching mode M_2 where M_1, $M_2 \in \{no\_caching$, $single\_cycle$, $multi\_cycle\}$ as follows:

$$d_{M_1}^{M_2} = cost_{M_1} - cost_{M_2}.$$

We say that a caching mode M_2 is better than a caching mode M_1 iff $d_{M_1}^{M_2} > 0$.

In the following sections, we examine the benefits of two caching modes, single-cyle caching mode and multi-cycle in relation to $no\_caching$ mode in agent programs. In particular, we specify in which conditions caching bring benefits to agent programs and quantify the improvement in performance. We also analyse worst-case senarios where caching might potentially make the query answering in agent programs slower.

5.3.2 Single-Cycle Query Caching

If the caching is done only within each single query-update cycle , we refer this mode of caching to the *single-cycle caching mode*. In the beginning of each query-update cycle, the cache is emptied, i.e., cached results are only kept within one query-update cycle. Alternatively, the cache can be emptied after the query phase, and this does not affect the abstract performance model given below. The average time cost per query-update cycle with query caching done for each single cycle is:

$$cost_{single\_cycle} = K \times (c_q + c_{ins}) + N \times c_{hit} + U \times c_u. \tag{5.2}$$

In Equation 5.2, for all N queries in a cycle, a cache lookup is performed to check if the query has already been cached, hence takes $N \times C_{hit}$. If it is a cache hit (i.e., the query is cached) then the cached answer is returned. Otherwise, a normal query is sent to the KRT and the returned answer is inserted into the cache; this requires $K \times (C_q + C_{ins})$. The total update time is as in the case of no caching (see Equation 5.1). In pratice, if the cache is implemented using a hash table, one can achieve C_{ins} and C_{hit} in constant time.

Clearly, single-cycle caching benefits an agent program when $d^{single\_cycle}_{no\_caching} > 0$, which is equivalent to:

$$(N \times c_q + U \times c_u) - (K \times (c_q + c_{ins}) + N \times c_{hit} + U \times c_u) > 0,$$

which is then equivalent to:

$$(N - K) \times c_q - K \times c_{ins} - N \times c_{hit} > 0.$$

As one can see, the single-cycle is beneficial to an agent program iff $(N - K) \times c_q > K \times c_{ins} + N \times c_{hit}$. The worst case happens when all queries are unique (i.e., $N = K$), and hence $d^{single\_cycle}_{no\_caching} = -K \times c_{ins} - N \times_c hit < 0$, the *single_cycle* caching mode becomes slower than *no_caching* mode by an amount of $N \times (c_{ins} + c_{hit})$. In other words, in the worst case, for each query, all three operations (e.g., cache lookup, KRT query, and cache insertion) have to be done.

5.3.3 Multi-Cycle Query Caching

The single-cycle caching mode, as described in the previous sub-section, clears the cache in the beginning of each query-update cycle. However, this may be inefficient because there is a high percentage of queries which have already been cached in the previous cycle according to Observation 5.2. The idea of the multi-cycle caching mode is to keep the cached results as long as possible, i.e., until they become invalid. To do so, the caching is performed in two steps, as illustrated in Algorithm 5.1.

Algorithm 5.1 Multi-Cycle Caching

```
% **Step1:** Query Phase
for each $query\ Q_i$ do
    $answer \leftarrow$ lookup$(Q_i, cache)$
    if $answer \neq null$ then
        return $answer$
    else
        $answer \leftarrow$ query$(Q_i, database)$
        put$(Q_i, answer, cache)$
        return $answer$
    end if
end for
% **Step2:** Update Phase
$queries \leftarrow \emptyset$
for each $update\ U_i$ do
    update$(U_i, database)$
    $queries \leftarrow queries \cup$ invalidate$(U_i, cache)$
end for
for each $query\ Q_j \in queries$ do
    delete$(Q_j, cache)$
end for
```

The first step is similar to single-cycle caching and occurs in the *query phase* of the cycle. In this step, all cache lookups are performed. If it is a cache hit, the query answer is returned. Otherwise, a query to KRT is performed, and the answer is returned to the agent program. The second step is done in the *update phase* of the cycle. Recall that in the update phase, the agent databases change, and hence some cached results become invalid. Therefore, it is necessary to eliminate these affected cached results from the cache. To do so, we need a mechanism to keep track of which cached results are affected by an update. This is based on the idea of belief revision presented in Section 4.6, where some beliefs are retracted/unbelieved because of an update in the database. This is where we can employ a truth maintenance system, i.e., to keep track of which beliefs should become invalid when an update occurs. Before looking at how to use such a TMS for that purpose, we will first see how the multi-cyle caching mode can improve agents' query answering process.

To be able to maintain a cache of queries over multiple cycles, we need to quantify following operations:

- $c_{invalid}$ is the average cost to retrieve which cached queries are affected by an update.
- c_{del} is the average cost to delete a query from the cache when it becomes invalid. As a cache can be implemented using a hash table, c_{del} is of constant time.

Note that for each query phase, we have in average $p \times N$ cached queries from the previous cycle and $(1 - p) \times N$ uncached. However, this does not take into account the number of queries cached within a cycle. If we assume K unique queries are distributed uniformly over cached and uncached queries, the number of uncached queries per cycle will become $(1 - p) \times K$. That is, $(1 - p) \times (N - K)$ is the number of queries which are not cached in the previous cycles, but are cached within the current cycle. Hence, the total time in query phase according to this model is:

$$(1 - p) \times K \times (c_q + c_{ins}) + N \times c_{hit} \tag{5.3}$$

For the time spent on update phase, we need to consider two steps. The first step is the cost of updating and retrieving invalidated queries. These need to be done for each update, and hence the time spent in this step is $U \times (c_u + c_{invalid})$. The other step is to delete invalidated queries from the current cache. Let us denote the total number of invalidated (unique) queries as $N_{invalid}$. In the best case where no cached query is invalid, $N_{invalid}$ is 0. In the worst case when all cached queries are removed, this number is the total number of cached entries. Here we take $N_{invalid}$ to be $(1-p) \times K$, i.e., the average number of uncached queries per cycle. We then have the total time in update phase according to this model:

$$U \times (c_u + c_{invalid}) + N_{invalid} \times c_{del} \tag{5.4}$$

The average time cost per query-update cycle with query caching done over multiple cycles is:

$$cost_{mult\_cycle} = (1-p) \times K \times (c_q + c_{ins}) + N \times c_{hit} + U \times (c_u + c_{invalid}) + N_{invalid} \times c_{del} \tag{5.5}$$

Multi-cycle caching mode will benefit an agent program if $d^{multi\_cycle}_{no\_caching} > 0$. From Equations 5.1 and 5.5, this is equivalent to:

$$N \times c_q + U \times c_u - ((1-p) \times K \times (c_q + c_{ins}) + N \times c_{hit} + U \times (c_u + c_{invalid}) + N_{invalid} \times c_{del}) > 0$$

which is equivalent to:

$$(\frac{N}{K} - 1) \times c_q + p \times (c_q + c_{ins}) > \frac{N}{K} \times c_{hit} + c_{ins} + \frac{U}{K} \times c_{invalid} + (1-p) \times c_{del}.$$

As the cache can be implemented using a hash table, the cost for lookup, insertion, and deletion can be of constant time. Also, timing results show that c_q is much higher than c_{hit}, c_{ins}, and c_{del}. Therefore, the performance of the multi-cycle caching mode depends mainly on N/K, p, and $\frac{U}{K} \times c_{invalid}$. The more queries repeated within a cycle (i.e., the larger $\frac{N}{K}$ is), the more queries repeated over multiple cycles (i.e., p is higher), and the smaller $\frac{U}{K} \times c_{invalid}$, the better the multi-cycle caching mode compared to no-caching mode. All three conditions (i.e., $\frac{N}{K} > 1$, $p > 50\%$, and $\frac{U}{K} \ll 1$) are satisfied given Observations 5.1, 5.2, and 5.3.

One might also wonder under which conditions multi-cycle caching outperforms single-cycle caching and whether it is possible to switch between caching modes to optimise the benefits of caching (if any) in agent programs. The answer to the first question is when $d^{multi\_cycle}_{single\_cycle} > 0$. Given $cost_{single\_cycle}$ from Equation 5.2 and $cost_{multi\_cycle}$ from Equation 5.3, we have the following condition under which multi-cycle caching will outperform single-cycle caching mode:

$$p \times K \times (c_q + c_{ins}) - U \times c_{invalid} - N_{invalid} \times c_{del} > 0. \tag{5.6}$$

If we replace $N_{invalid}$ by $(1 - p) \times K$ then Condition 5.6 is equivalent to:

$$p \times K \times (c_q + c_{ins}) - U \times c_{invalid} - (1 - p) \times K \times c_{del} > 0$$

which is then equivalent to:

$$p \times K \times (c_q + c_{ins}) > U \times c_{invalid} + (1 - p) \times K \times c_{del}. \tag{5.7}$$

Because the cache is implemented using a hash table, it is reasonable to assume $c_{del} = c_{ins}$. Thus, $p \times K \times c_{ins} > (1 - p) \times K \times c_{del}$ when $p > 1 - p$. Moreover, recall that from Observation 5.2 and Observation 5.3 in Section 5.2.3, p ranges from 52% to over 90% (i.e., $p > 1 - p$) and $K \gg U$ in all combinations of agent platforms/environments/programs in the experiment.This means that the Condition 5.7 can be satisfied, and hence the performance of agent program with multi-cycle caching mode will be better than one with single-cycle caching mode. However, note that we need to assume that the cost to get which queries become invalidated after an update, i.e., $c_{invalid}$, is not much greater than the cost to perform a query, i.e., c_q. In the next section, we will explain why it is reasonable to make this assumption.

5.4 Truth Maintenance for Multi-Cycle Query Caching

In this section, we present an approach to maintain sets of queries that are affected by database updates using a lightweight truth maintenance system. In particular, we do not cache all inferences as in the conventional TMS. Instead, we only keep track of queries (datum nodes) and the set of ground facts (assumptions) used to derive them.

In Algorithm 5.1, the key factor to allow query caching over multiple query-update cycles is the function invalidate which returns a list of queries becoming invalidated (i.e., incorrect) after an update. For example, assumed that an agent program includes a knowledge-base as follows (the example is from the book "the Art of Prolog" [87]).

```
father(abraham,isaac).     male(isaac)
father(haran,lot).         male(lot)
father(haran,milcah).      female(milcah).
father(haran,yiscah).      female(yiscah).
son(X,Y):-father(Y,X), male(X).
```

LISTING 5.6: A simple agent's knowledge-base in PROLOG

If a query `Q=son(haran,X)` is asked, the search tree as in Figure 5.2 will be made. The answer to this query against the current agent database is `{S=lot}`, i.e., a mapping from `S` to `lot`, is added into the cache. By looking at the search tree, we

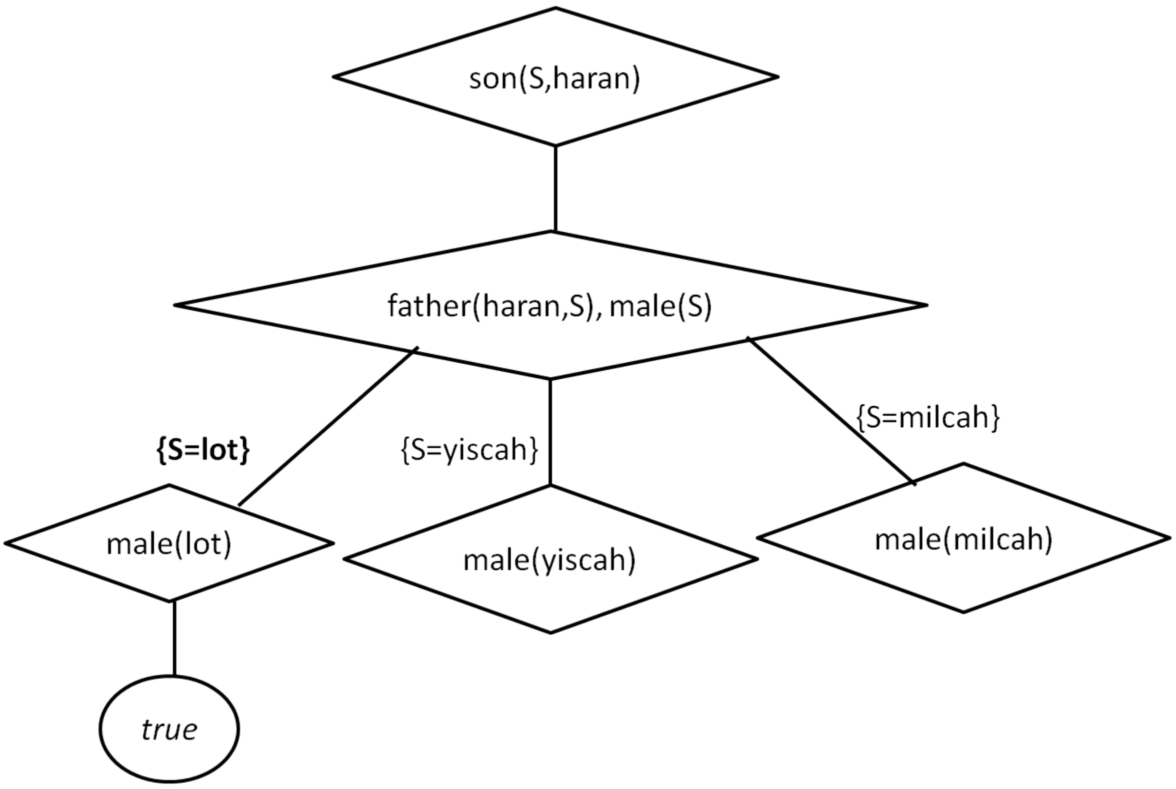

FIGURE 5.2: A simple search tree from query $son(haran, X)$ from [87].

can see that query `son(haran,X)` depends on `{male(lot),father(haran,lot)}`. In other words, we call that `{male(lot),father(haran,lot)}` is the support set of Q, following Definition 5.2. If in the update phase of a later cycle there is an update such that `male(lot)` is deleted from the database, then the answer `{S=lot}` is no longer valid, and should be removed from the cache.

Definition 5.2 (Support Set of a Query)**.** A support set of a query Q against a knowledge base KB is the set of explicit ground facts $S(Q) = \{f \mid f \in KB\}$ where each f is used to find a solution to Q, i.e., $\{f\} \cup KB' \models \delta(Q)$ where $KB' \subset KB \wedge KB' \not\models \delta(Q)$ and $\delta(Q)$ is an instantiation of Q. If there is no solution to q then $S(Q) = \emptyset$.

The idea is now to maintain a database of support sets for all queries so far. From this database, one can compute which queries are possibly affected by an element of a support set. We refer to this set of queries as *the invalidated set* of a

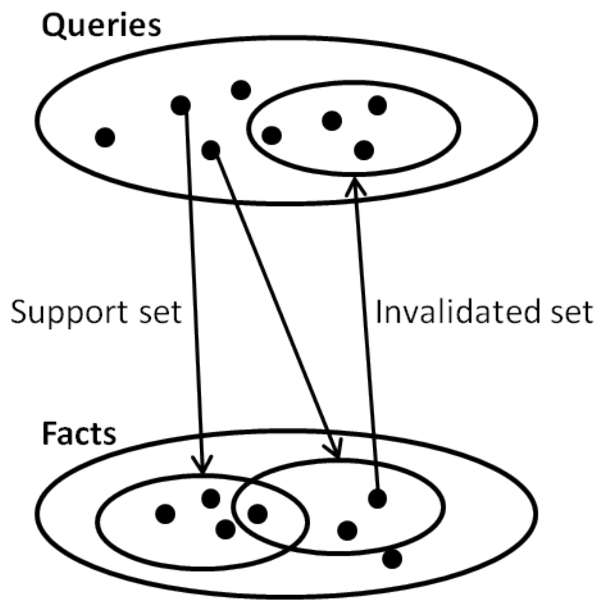

FIGURE 5.3: Mappings from queries to their support sets and from facts to their invalidated sets.

fact. The mappings from queries and facts to their corresponding support sets and invalidated set respectively are illustrated in Figure 5.3.

From the example in Listing 5.6, one can compute the following support sets based on the queries $son(haran, Y)$ and $son(X, Y)$.

SupportSet($son(haran, Y)$)=$\{father(haran, lot), male(lot)\}$

SupportSet($son(X, Y)$)=$\{father(abraham, isaac), male(isaac), father(haran, lot), male(lot)\}$

Then we will have some invalidated sets as follows:

$\ldots$

InvalidatedSet($male(lot)$)=$\{son(haran, Y), son(X, Y)\}$

InvalidatedSet($father(abraham, isaac)$)=$\{son(X, Y)\}$

$\ldots$

From a TMS viewpoint, a support set of a query is the set of assumptions where the instantiation of the query holds and an instantiation of a query is a datum in the dependency graph. This is demonstrated in Figure 5.4. However, instead of

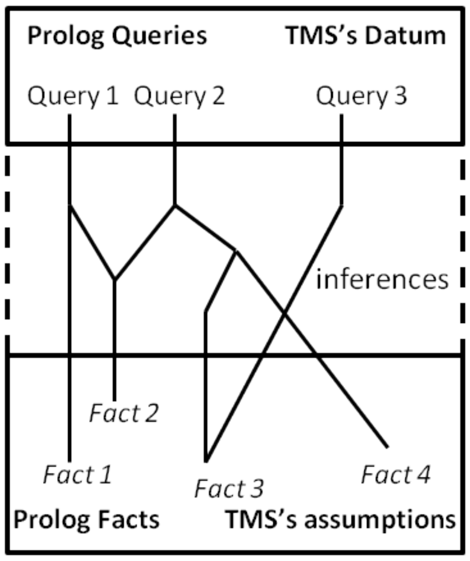

FIGURE 5.4: Prolog queries and facts in correspondence to TMS datum nodes and assumptions.

recording all intermediate inferences as justifications as in a conventional JTMS, here we only record the relationship between assumptions and data representing queries. All intermediate inferences (inferences in the dashed area of Figure 5.4) are omitted, and hence the cost of maintaining a full dependency graph is significantly reduced.

Obviously, this approach is not fine-grained in the sense that the support sets of sub-goals, e.g., in the running example $father(X, Y)$ is a sub-goal of $son(X, Y)$, will not be computed. However, note that although the knowledge base is growing non-monotonically (i.e., $KB \models Q$ does not imply $KB' \models Q$ where $KB \subset KB'$), the dependency graph is not, i.e., all justifications and beliefs are never removed from the graph. Therefore, this leads to increasing complexity for the implementation of query caching if a full JTMS is implemented. In fact, Forbus and de Kleer reported in [38] that it is usually a bad idea to connect a full TMS to a PROLOG interpreter because of cost of keeping and maintaining the cache of all inferences is not cheaper than just rerunning the inference rules. What we do here based on the observations that there are substantial number of queries are repeated and caching the queries' answers will make the agent program more efficient. Thus we only need to keep track of the original facts (assumptions), not the intermediate steps, to derive a query answer.

5.5 Experimental Results

In this section we present an implementation of the caching models described earlier for the GOAL agent programming language with SWI-Prolog as the KRT. Both single- and multi-cycle caching were implemented. The implementation of single-cycle caching is straightforward, as described in Sub-section 5.3.2. In what follows, we focus on the implementation of multi-cycle caching. The multi-cycle caching implementation follows Algorithm 5.1. To implement the `invalidate` operation, we used a meta-interpreter written in Prolog that, in addition to the answer to a query, returns the ground facts used to answer the query. Calls to SWI-Prolog are replaced by calls to the meta-interpreter. Apart from providing the ground facts supporting a query, the meta-interpreter does not change the result of the original query.

The answer to each query is stored in a hash table *queryCache*. Each ground fact f returned by the meta-interpreter is also stored together with the set of queries it may invalidate, *invalidates(f)* in a hash table. In later query-update cycles, if an update (insertion or deletion) of f is performed then, for each query in *invalidates(f)*, its cached result is removed from *queryCache*, and f is also removed from *invalidates*. Note that this means c_q in Equation 5.5 includes also the cost of `invalidate`, and the `invalidate` operation now becomes only a hash table lookup to retrieve invalidated queries. The computation of dependency information is performed at run-time rather than compile-time, which is useful to switch between caching modes depending on different agent programs.

To measure the benefits of different query caching modes, we run the extended GOAL version which allows users to choose different types of caching modes in two classic problems in agent programming, the Blocks World problem and the elevator problem. Blocks World, introduced in [92], is an environment consisting of a table and a set of blocks. A goal is a state of the world where the blocks are put on top of another to build one or more towers. This is a classic planning problem where the initial state and the goal state are clearly specified, the agent has the full control, and the environment is fully observable and deterministic. The size of a Blocks World problem is determined by the number of blocks in the environment. The Elevator problem, on the other hand, is a dynamic environment which contains a set of elevators which are controlled by different agents. A simulator randomly generates people and their actions such as calling for an elevator, entering/leaving an elevator, or going to a specific floor. Because each agent can only observe its own elevator and the simulation of people's actions is random, this environment is only partly-observable and non-deterministic. The size of this problem is the number of floors.

Table 5.1 shows the comparison between different caching models in GOAL where h represents the percentage of cache hit . The figures reported are the average of 5 runs and timing is in microseconds. Although the log-files show that the average query times for calls to the meta-interpreter are about 1.5 to 2 times higher than normal queries, as the cache is cleared less often, the number of calls to SWI-Prolog decreases resulting in a reduction in average query times compared to single-cycle caching.

Problem	Caching	h	c_q	c_u
Blocksworld10	No	0%	53.83	52.67
Blocksworld10	Single-cycle	27%	44.10	46.42
Blocksworld10	Multi-cycle	36%	43.74	40.48
Blocksworld50	No	0%	42.04	44.89
Blocksworld50	Single-cycle	32%	38.86	43.68
Blocksworld50	Multi-cycle	51%	31.79	37.35
Blocksworld100	No	0%	37.07	41.63
Blocksworld100	Single-cycle	31%	33.03	42.99
Blocksworld100	Multi-cycle	54%	30.21	37.90
Elevator10	No	0%	19.15	19.87
Elevator10	Single-cycle	83%	3.40	20.52
Elevator10	Multi-cycle	90%	2.87	20.15
Elevator50	No	0%	19.81	19.21
Elevator50	Single-cycle	65%	7.37	20.00
Elevator50	Multi-cycle	79%	5.80	17.81
Elevator100	No	0%	20.23	19.12
Elevator100	Single-cycle	65%	7.61	19.92
Elevator100	Multi-cycle	77%	6.10	18.21

TABLE 5.1: Comparison of different caching modes

5.6 Related Work

The idea of caching query calls and answers in deductive reasoning systems had been proposed in Tabled Logic Programming [89]. The idea underlying tabling is that sub-goals and their (possibly incomplete) answers are stored during searching for an answer to a query. The main goal of this technique is to maintain termination of logic programs. However, in many Prolog systems, tables are cleared after each top-level query (e.g., the main goal). XSB-Prolog as mentioned in [89] provides support to maintain tables of dynamic predicates when an update occurs with a technique called *incremental tabling*, which is related to the field of truth maintenance. In all cases the tabled predicates need to be pre-declared in the logic program.

Maintaining cached results after updates is also similar to the problem of *incremental view maintenance* in database systems [44]. Provably efficient algorithms to find minimised incremental changes in relational databases exist, e.g., [42]. In ontology systems, there is also work on caching ontological inferences and performing updates on top of the cached inferences instead of recomputing the inferences from scratch, see [77]. The most relevant work to the approach presented in this chapter is the one reported in [2]. In fact, the argument that caching can improve the performance of agent programs based on the observations in [2]. We also used the abstract performance model for single-cycle query caching from this work.

5.7 Conclusion

In this chapter, we presented an approach to query caching in agent programming using a lightweight TMS as a means of dependency tracking. Firstly, we analysed the query and update phases of an agent's query-update cycle, with the GOAL agent programming language as an example. Next, we quantified the benefits of query caching by extending the abstract performance model given in [2]. Specifically, we showed that according to the observations in [2], it can be more efficient for agent programs to implement multi-cycle caching. An approach to implement query caching in GOAL with SWI-Prolog as the knowledge representation technology was also described, with experimental results showing that query-caching, especially in multi-cycle caching mode, can make agent programs more efficient by reducing time for re-querying the KRT.

Chapter 6

Detecting Geospatial Ontology Mapping Errors

6.1 Introduction

Nowadays, the process of creating geospatial data involves not only expert modellers but also dedicated voluteers. This trend brings the advantages of both sources of data to recently developed geospatial databases: the authoritative, consistent, and standardised data from experts and the more up-to-date and feature-rich information from the community. As an example, [34] use the data from Ordnance Survey, the UK's national mapping agency, and from Open Street Map, a free open-sourced map which allows collaboration in creating and editing maps, to investigate the methodologies to link geospatial data from two separate sources to take the advantages of both.

One problem with linking data from different sources, especially automatic data linking, is to maintain the consistency of data. This is equivalent to the problem of finding the potential errors in auto-generated mappings, as incorrect mappings can lead to the global system inconsistency.

In this chapter, we present an approach to detect such mapping errors in a geospatial knowledge-based system using an ATMS. We focus on the problem of finding ontology-mappings responsible for contradictions in a geospatial Knowledge-Base (KB) generated by ontology mappers[1]. As an example, we consider the KB using *the Logic of NEAR and FAR* (LNF) introduced in [33]. We also configure the ATMS to meet the problem requirements.

The research questions, objectives, and the contributions of the work presented in this chapter are as follows.

Research Questions How can an ATMS be constructed to solve the ontology debugging (axiom pinpointing) problem? Is it feasible for an ATMS to find all minimal explanations in a reasonable amount of time?

Research Objectives

1. To show a use case where an ATMS can be employed to detect mapping errors between two geospatial ontologies.
2. To construct a general framework which use an ATMS to give minimal explanations for inconsistency derived after combining two geospatial ontologies and the mappings.
3. To implement the ATMS and to verify that the system can produce all mapping errors (minimal explanations) within a reasonable amount of time with a realistic dataset.

Contributions We show that a "classic" ATMS can be constructed to solve the problem of ontology debugging (axiom pinpointing) in a rule-based system consisting of Horn-like rules. We also show that the framework can be used to detect all mappings errors in the use case of Nottingham city

[1]An ontology mapper generates mappings of instances or concepts between multiple ontologies.

centre where individuals of the Open Street Map ontology are mapped to an individual in the Ordnance Survey ontology.

The structure of this chapter is as follows. In Section 6.2, we introduce the problem of finding errors in the mappings of two geospatial data sets which are generated using a qualitative approach, namely the Logic of NEAR and FAR. Section 6.3 shows how to detect all incorrect mappings using an ATMS. We then present the algorithms for maintaining node labels' properties mentioned in the previous chapter, which are implemented for this particular problem, in Section 6.4. The algorithms are based on ones given in [25]. Finally, in the last two sections, we give correctness proofs and experimental results of the system based on the geospatial dataset of Nottingham City Centre.

6.2 Finding Incorrect Mappings in a Geospatial Knowedge-based System using the Logic of NEAR and FAR

In [33], the authors use a fragment of LNF to detect incorrect instance-matchings[2] generated from two different geospatial ontologies. In particular, they used the data sets from Open Street Map (OSM)[3] and Ordnance Survey of Great Britain (OSGB)[4] to generate mappings of geospatial objects from these two sources using some criteria. A mapping of two objects is of the following form, where X and Y are the ids of two geospatial objects from OSM and OSGB respectively:

$$OSM : X = OSGB : Y$$

[2]A mapping between two sources of data (e.g., geospatial ontologies) can be at two levels: the concept level and the instance level.
[3]http://www.cs.nott.ac.uk/ hxd/evaluation/OpenStreetMap.owl
[4]http://www.ordnancesurvey.co.uk/ontology/BuildingsAndPlaces/v1.1/BuildingsAndPlaces.owl

However, these mappings might not always be correct, and there exists a need to check whether there is any inconsistence in a set of generated mappings. To do so, they introduced the Logic of NEAR and FAR. In general, the Knowledge-Base contains a set of rules R and a set of facts F, as in [33]. The facts are binary ground formulas of the forms $BEQ(a, b)$, $NEAR(a, b)$, and $FAR(a, b)$ where $a,\ b$ are geospatial objects and $BEQ, NEAR, FAR$ are binary predicates representing the fact that two objects are considered to be possibly in the same location, nearby, or far from each other respectively. For $\delta = 20m$, $BEQ(a, b)$ is generated if a and b are within a distance d and $d \leq \delta$. Similarly, $NEAR(a, b)$ and $FAR(a, b)$ are generated if $\delta < d \leq 2 * \delta$ and $d > 4 * \delta$, respectively. These facts are generated using the geospatial data (i.e., locations) of all objects from the original sources (e.g., OSM and OSGB).

The LNF rules are rules of the form $A \rightarrow B$. These rules are introduced in [33], where $BEQ, FAR, NEAR$ are binary predicates and a, b, c, d, e are variables:

Rule 1 $BEQ(a, a)$;

Rule 2 $BEQ(a, b) \rightarrow BEQ(b, a)$;

Rule 3 $NEAR(a, b) \rightarrow NEAR(b, a)$;

Rule 4 $FAR(a, b) \rightarrow FAR(b, a)$;

Rule 5 $BEQ(a, b) \wedge BEQ(b, c) \rightarrow NEAR(a, c)$;

Rule 6 $BEQ(a, b) \wedge NEAR(b, c) \wedge BEQ(c, d) \rightarrow \neg FAR(d, a)$;

Rule 7 $NEAR(a, b) \wedge NEAR(b, c) \rightarrow \neg FAR(a, c)$;

Rule 8 $BEQ(a, b) \wedge BEQ(b, c) \wedge NEAR(c, d) \rightarrow \neg FAR(d, a)$;

Rule 9 $BEQ(a, b) \rightarrow NEAR(a, b)$;

Rule 10 $FAR(a, b) \rightarrow \neg NEAR(a, b)$;

Rule 11 $BEQ(a,b) \wedge FAR(b,c) \rightarrow \neg NEAR(c,a)$;

Rule 12 $BEQ(a,b) \rightarrow \neg FAR(a,b)$;

Rule 13 $BEQ(a,b) \wedge BEQ(b,c) \rightarrow \neg FAR(c,a)$;

Rule 14 $BEQ(a,b) \wedge BEQ(b,c) \wedge BEQ(c,d) \rightarrow \neg FAR(d,a)$; and

Rule 15 $BEQ(a,b) \wedge BEQ(b,c) \wedge BEQ(c,d) \wedge BEQ(d,e) \rightarrow \neg FAR(e,a)$.

Apart from **Rule 1**, which means that an object is always within a distance δ of itself, other rules are self-explanatory. **Rule 7** is only applicable for points, not polygon objects, and hence mappings of polygon objects needs to take this into account and remove **Rule 7**.

Each mapping $OSM : X = OSGB : Y$ makes two objects $OSM : X$ and $OSGB : Y$ equivalent in the KB. For example, if $NEAR(OSM : X, OSM : Z)$ and there exists a mapping $OSM : X = OSGB : Y$, then $NEAR(OSM : Y, OSM : Z)$ is also in the KB. As there is only three predicates in the logic, to implement equality of objects, we can encode the mappings as the following additional rules:

Rule 16 $a = b \rightarrow b = a$ (mappings are symmetrical);

Rule 17 $a = b \wedge BEQ(a,c) \rightarrow BEQ(b,c)$;

Rule 18 $a = b \wedge NEAR(a,c) \rightarrow NEAR(b,c)$;

Rule 19 $a = b \wedge FAR(a,c) \rightarrow FAR(b,c)$;

Rule 20 $a = b \wedge \neg BEQ(a,c) \rightarrow \neg BEQ(b,c)$;

Rule 21 $a = b \wedge \neg NEAR(a,c) \rightarrow \neg NEAR(b,c)$;

Rule 22 $a = b \wedge \neg FAR(a,c) \rightarrow \neg FAR(b,c)$;

The task required is twofold. The first part is to check whether there are any inconsistencies under the Logic of NEAR and FAR specified by the above rules, given generated mappings. Secondly, if there is an inconsistency then find minimal sets of mappings responsible for the inconsistency.

6.3 The ATMS-based Approach to Mapping Errors-Detection

To solve the problem presented in Section 6.2, we use an ATMS introduced in Section 4.5. The whole framework is illustrated in Figure 6.1. We use a reasoner to reason under the Logic of NEAR and FAR, which includes the rules in Section 6.2. To introduce inconsistency, we have an additional rule stating that a fact and its negation cause a contradiction:

$\perp$-rule $A(a, b) \wedge \neg A(a, b) \rightarrow \perp$ where $A \in \{BEQ, NEAR, FAR\}$.

There is a clear mapping between the problem of detecting errors and the operations of the ATMS. In particular, all generated BEQ, $NEAR$, and FAR facts and LNF rules can be encoded as ATMS premises and each mapping is represented by an ATMS assumption. We then have a reasoner to infer new facts as well as to discover inconsistency. The job of the ATMS is to maintain the cache of inferences in its dependency graph and compute all possible derrivations of a node, including $n_{\perp}$, in each node's label.

The system operates in a cycle. At each cycle the reasoner applies an inference rule to a set of facts which are not currently known to be inconsistent and sends the inference to the ATMS in form of a justification if such a justification does not exist. The ATMS creates nodes and updates the dependency graph between nodes using the justification. In addition, the ATMS also maintains consistency,

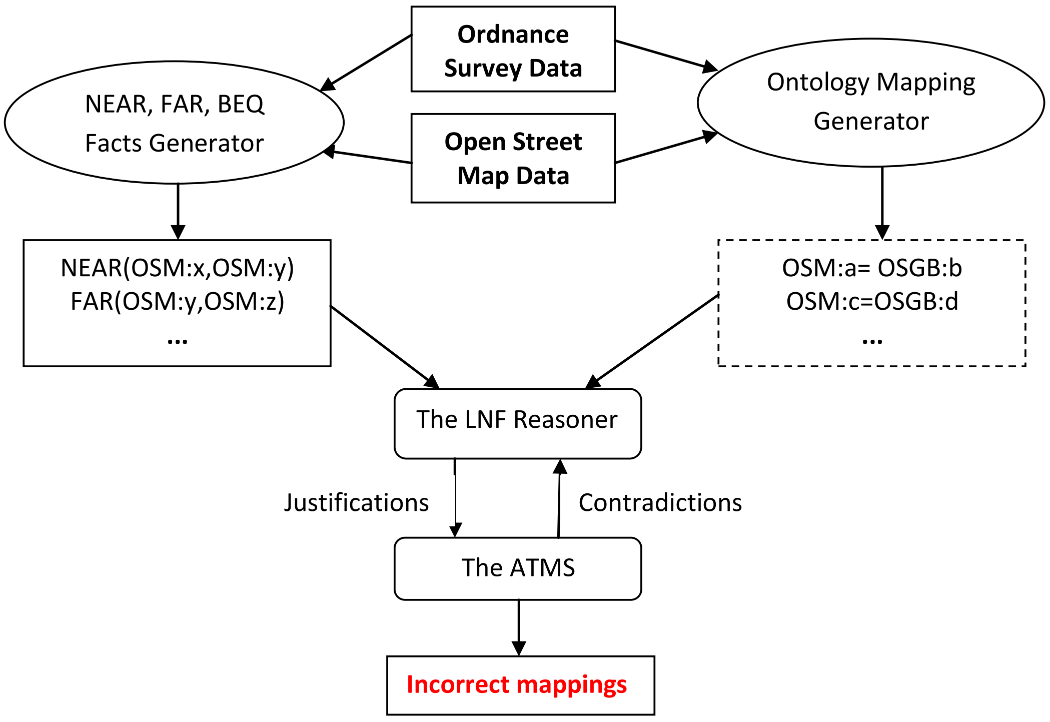

FIGURE 6.1: The Framework to Find Incorrect Mappings of 2 Geospatial Ontologies, Open Street Map (OSM) and Ordnance Survey Great Britain (OSGB).

minimality, soundness, and completeness of each node's label using the *label update propagation* algorithms mentioned in Section 4.5.3. The reasoner keeps making inferences until no inference rule can be applied. At this point, each environment in the label of a node n_{datum} is a minimal set of axioms that can used to derive *datum*, and the label of $n_{\perp}$ consists of sets of mappings responsible for inconsistency.

6.4 Algorithms for Label Update Propagation in the ATMS

This section describes the algorithms used to update node labels in the ATMS. The algorithms are based on ones in [25]. When the ATMS receives a new justification $J : x_1, \ldots, x_k \Rightarrow n$, it invokes PROPAGATE$(J, a, I)$ to update node n's label and propagate the changes to other nodes in the ATMS. PROPAGATE takes three parameters: $J : x_1, \ldots, x_k \Rightarrow n$ is the justification for the node whose label to be updated, a is an antecedent of J whose label has been updated, and I is the newly added environments. With a new justification J sent from the reasoner, I and a are given as $\{\{\}\}$ and $\emptyset$ respectively. PROPAGATE first computes the label update for node n by calling COMPUTE-SINGLE-LABEL-UPDATE. If there exists a non-empty update L, it updates n's label with the environments in L by calling UPDATE-NODE-LABEL.

Algorithm 6.1 Propagate incremental label update

```
procedure PROPAGATE($J : x_1, \ldots, x_k \Rightarrow n, a, I$)
    $L \leftarrow$ COMPUTE-SINGLE-LABEL-UPDATE($a, I, \{x_1, \ldots, x_k\}$)
    if $L \neq \{\}$ then
        UPDATE-NODE-LABEL($L, n$)
    end if
end procedure
```

Procedure COMPUTE-SINGLE-LABEL-UPDATE computes the label update for a node n, when a is a member of n's antecedents, I is a set of new environments recently added to node a, and X is the set of J's antecedents (i.e., $\{x_1, \ldots, x_k\}$) in Algorithm 6.1).

Algorithm 6.2 Compute single label update

```
procedure COMPUTE-SINGLE-LABEL-UPDATE($a, I, X$)
    for all $h \in X, h \neq a$ do
        $I' \leftarrow \{e \cup e' \mid e \in I, e' \in \mathit{label}(h)\}$
        $I \leftarrow I' \setminus \{e \mid e' \subseteq e, e_\perp \subseteq e, e \in I', e' \in I', e_\perp \in \mathit{label}(n_\perp)\}$
    end for
    return $I$
end procedure
```

Procedure UPDATE-NODE-LABEL updates the label of n and propagates the changes, i.e., L, to its consequences if n is not a contradictory node. It also maintains soundness and minimality properties of nodes' label. For clarity and consistency between different chapters in this thesis, this procedure is slightly different from the original one from [25] as it does not remove subsumption and nogoods from the label update L after propagating the label update. However, this does not affect the performance of the label update propagation significantly.

Algorithm 6.3 Update the current label and propagate to consequences

```
procedure UPDATE-NODE-LABEL($L, n$ )
    if $n = n_\perp$ then
        for all $e \in L$ do
            UPDATE-FALSITY($e$)
        end for
    else
        $L \leftarrow L \setminus \{e \mid e_n \subseteq e, e \in L, e_n \in label(n)\}$
        $L_n \leftarrow label(n) \setminus \{e_n \mid e \subseteq e_n, e \in L, e_n \in label(n)\}$
        $label(n) \leftarrow L_n \cup L$
        if $L = \{\}$ then
            return
        end if
        for all $J$ where $n \in antecedents(J)$ do
            PROPAGATE($J, n, L$)
        end for
    end if
end procedure
```

Procedure UPDATE-FALSITY updates the nogood database in the label of $n_\perp$

when a new nogood is found. It also triggers label update in all nodes to remove nogoods and its supersets from each label.

Algorithm 6.4 Update nodes when a nogood e is found

```
procedure UPDATE-FALSITY($e$)
    Add $e$ into $n_\perp$
    for all node $n \neq n_\perp$ in the ATMS do
        $L' = \{e' \mid e_\perp \subseteq e', e' \in label(n), e_\perp \in label(n_\perp)\}$
        $label(n) \leftarrow label(n) \setminus L'$
    end for
end procedure
```

6.5 Correctness

This section is to show that the approach works correctly, i.e., the label of a datum node is correct, complete, consistent, and minimal (see Section 4.5.2). Before proving that the properties of node's label hold, we firstly show that the reasoner logic (LNF) of the reasoner is sound and complete. This has been proved in [33]. Secondly, we need to show that the reasoner is terminating. This can be established from the facts that the reasoner is forward-chaining and the inference rules (LNF rules) do not contain loops. The following theorems show that the four properties of a node label are maintained.

Theorem 6.1 (Label soundness)**.** *For each datum node n and each environment in its label, there is a sequence of rule applications produced by the reasoner, such that the only assumptions used in the derivation are in the label's environment.*

Proof. The proof is by induction on the longest chain of justifications connecting the datum to assumptions. Environments are added when the ATMS receives a justification from the reasoner (see Algorithm 6.1). If the antecedents of the justification are assumption nodes or premise nodes, the theorem follows immediately because we have a one step derivation of the datum from those assumptions or premises. The inductive step is as follows. After the environment which reflects the assumptions used in the derivation is added to the datum's label, it is updated to maintain soundness. In particular, if an environment in the label of n is discovered to be a superset of another environment in the label of n (i.e., the label is non-minimal) or an environment in the label of $n_\perp$ (i.e., the label is inconsistent), it is removed (see Algorithm 6.2). This step, i.e., maintaining label minimality and consistency, does not change any environment (e.g., no assumption is added or removed from an environment), and hence soundness is maintained. □

Theorem 6.2 (Label completeness relative to reasoner)**.** *Every set of assumptions A from which a datum n can be derived given the set of justifications produced by the reasoner so far, is a superset of some environment in the datum's label.*

Proof. This theorem is proved by induction on the length of the derivation (the length of the chain of justifications produced by the reasoner). For a one step derivation (where the corresponding justification for n has assumption nodes or premise nodes as ancetedents) this is immediate. For a k step derivation, assume that the justification for n has n_1 and n_2 as antecedents, and that all known derivations for n_1 use one of the sets of assumptions $e_1, \ldots, e_m$ and for n_2, $e'_1, \ldots, e'_{m'}$. Since n_1 and n_2 occur as the k-1st and k-2nd steps in the derivation of n, the inductive hypothesis applies (i.e., the labels of n_1 and n_2 are complete given the justifications produced so far). As in Algorithm 6.2, all ways to derive n use $e_1 \cup e'_1$ or $e_1 \cup e'_2$, …, or $e_m \cup e'_{m'}$. This set of environments will then be added to n's label by Algorithm 6.3. The 'superset' comes from the fact that Algorithms 6.2 and 6.3 check for subsumption of environments and remove the ones which contain redundant assumptions. □

Theorem 6.3 (Label Consistency)**.** *No environment in the datum node's label is inconsistent, i.e., is a superset of an environment where $n_\perp$ holds.*

Proof. If a $nogood$ environment of $n_\perp$ is found (see Algorithm 6.3), Algorithm 6.4 ensures any environment of a node in the ATMS, apart from $n_\perp$, is not a superset of the $nogood$ environment. □

Theorem 6.4 (Label minimality)**.** *No environment in the datum's label is a subset of any other.*

Proof. Guaranteed by subsumption tests and removal in Algorithms 6.2 and 6.3. □

6.6 Preliminary Experimental Results

In this section, we describe an implementation of an error detecting system that comprises a forward-chaining rule-based reasoner and an ATMS. The system, including the reasoner and the ATMS, is written in POP11.[5] The reasoner is a set of rules (**Rule 2-22**) from Section 6.2, except **Rule 7** as mentioned before. For **Rule 1**, the system generates all facts of the form $B(a, a)$ where a is a constant in the KB at compile-time. Other facts, e.g., instances of BEQ, $NEAR$ and FAR predicates, are generated from the location of each object and explicitly given to the reasoner. Rules and BEQ, $NEAR$, FAR facts are encoded as premises (i.e., always true). The mappings (e.g., $OSM : X = OSGB : Y$) are encoded as assumptions as these might or might not be correct.

[5]`http://www.cs.bham.ac.uk/research/projects/poplog/freepoplog.html`

The generation of mappings and original BEQ, $NEAR$, and FAR facts is as in [33]. BEQ, $NEAR$, and FAR facts are generated using Nottingham City Centre's geospatial data from OSGB and OSM map data. The mappings are generated using similarity in place names. The experiments were performed on a PC with dual core 2.2GHz Intel Pentium and 4GB RAM PC running Ubuntu 12.04. Initially, there were a total of 11557 original BEQ, $NEAR$, and FAR facts and 219 mappings. After termination, the reasoner and the ATMS detect 72 minimal sets of incorrect mappings. The total time for the reasoner to derive all facts and contradictions is 61.7 seconds while the ATMS spends 40.6 seconds to find all mappings leading to $n_{\perp}$. During its run, the ATMS has built a dependency graph of 15440 nodes and 30699 justifications. There are also 21198 justifications for $n_{\perp}$. It is interesting that although there are many justifications for $n_{\perp}$, the ATMS still performs well. The reason behind this is the average size of $n_{\perp}$'s environment, which is only one in this case. Note that the number of justifications as well as the average size of environments in $n_{\perp}$'s label will greatly affect the performance of the ATMS because of minimalisation process . In this case, because all LNF rules and facts are encoded as premises and mappings are encoded as assumptions, there is only one mapping (average) in each minimal set of mappings responsible for inconsistency, and hence the ATMS still performs well.

In Example 6.1, object $OSGB : 1000002309051190$ (*Castle Clinic*[6] located at a street called The Ropewalk) is mapped to object $OSM : 99999874$ (A pub named *The Ropewalk*[7]). This is then revealed as an incorrect mapping because while the names are similar, they are located far away from each other. To see why the mapping is incorrect, we also provide a facility to print a derivation of $\perp$. In Example 6.1, we also show two derivations of $\perp$. These derivations involve two

[6]Address: 18-20 The Ropewalk, Nottingham NG1 5DT
[7]Address: 107-111 Derby Rd, Nottingham NG1 5BB

steps (i.e., two justifications). $BEQ(OSM:99999874, OSM:99999874)$ is generated at compile-time and is a premise, and hence there is no justification for it. Note that although it is possible to have many justifications for a node, the nodes' labels are always kept minimalised due to the minimalisation process.

Example 6.1. *An example of minimal sets of incorrect mapping is:*

$$\{OSGB:1000002309051190 = OSM:99999874\}$$

Trace 1:

```
<\atms_justification: 63 >
 falsity
 DERIVED FROM
     [[BEQ OSM:99999874  OSM:99999874]
      [FAR OSM:99999874  OSM:99999874]
      [(=> (and (BEQ ?A ?B) (FAR ?A ?B)) (false))]]

<\atms_justification: 35 >
 [FAR OSM:99999874  OSM:99999874]
 DERIVED FROM
     [[SAMEAS OSGB:1000002309051190  OSM:99999874]
      [FAR OSGB:1000002309051190  OSM:99999874]
      [(=> (and  (SAMEAS ?X ?Y) (FAR ?X ?B)) (FAR ?Y ?B))]]
```

Trace 2:

```
<\atms_justification: 76 >
 falsity
 DERIVED FROM
** [[BEQ OSM:99999874 OSM:99999874]
```

```
      [BEQ OSM:99999874 OSM:99999874]
      [BEQ OSM:99999874 OSM:99999874]
      [BEQ OSM:99999874 OSM:99999874]
      [BEQ OSM:99999874 OSM:99999874]
      [(=> (and
                       (BEQ ?A ?B)
                       (BEQ ?B ?C)
                       (BEQ ?C ?D)
                       (BEQ ?D ?E)
                       (FAR ?E ?A))
           (false))]]

<\atms_justification: 35 >
 [FAR OSM:99999874  OSM:99999874]
 DERIVED FROM
     [[SAMEAS OSGB:1000002309051190  OSM:99999874]
      [FAR OSGB:1000002309051190  OSM:99999874]
      [(=> (and  (SAMEAS ?X ?Y) (FAR ?X ?B)) (FAR ?Y ?B))]]
```

6.7 Conclusion

In this chapter, we showed how to employ the original ATMS to solve a real-world problem, namely detecting errors in auto-generated mappings between two ontologies. In addition, correctness proofs of the algorithms (i.e., the ability to maintain correct ATMS's node labels) were also given. Finally, experimental results showed that ATMS can find all incorrect mappings of the given data-set within a reasonable amount of time (under one minute). As the problem presented in this chapter uses a logic supporting only Horn-clause inferences, we did not need to modify an ATMS significantly. In the next chapter, we will extend the current ATMS to deal with a reasoner of more expressive logic with disjunction and loops.

Chapter 7

Debugging Ontologies with Disjunctions and Loops

7.1 Introduction

In the previous chapter, an ATMS is applied to find incorrect auto-generated geospatial mappings. The inference rules in the reasoner are Horn-like rules, and hence using the ATMS is straightforward. In this chapter we show that an ATMS can also be used for axiom pinpointing, that is, finding the minimal set of axioms responsible for an unwanted consequence, even in a more expressive description logic such as $\mathcal{ALC}$, which allows disjunction constructs. More specifically, we present a system which returns all minimal sets of axioms responsible for the derivation of inconsistency in an $\mathcal{ALC}$ ontology (where all inclusion axioms have an atomic concept on the left). Following Sirin *et al* [86], we refer to these sets of axioms as *explanations*.

Our approach involves using a modified Assumption-Based Truth Maintenance System (ATMS) [23] to trace inferential dependencies between formulae and compute the minimal sets of ontology axioms responsible for a contradiction.

The research questions, objectives, and the contributions of the work presented in this chapter are as follows.

Research Questions How can a 'classic' ATMS (e.g., as described in [25]) be extended to support logics featuring disjunctions and cyclic definitions? How well the extended ATMS can perform in both synthetic and realistic ontologies?

Research Objectives

1. To define a variant of the $\mathcal{ALC}$ logic, which we call the dictionary $\mathcal{ALC}$, extending the unfoldable $\mathcal{ALC}$ terminologies with cyclic definitions.
2. To extend the data-structures of the original ATMS to support non-determinism, i.e., disjunctions, and cyclic definitions of concepts.
3. To implement the new ATMS, which we call the D-ATMS, and to conduct experiences comparing the performance of the system with other reasoners such as Pellet and MUPSter.

Contributions The main technical contribution of this work is extending the ATMS to deal with disjunctions and loops. The notion of an ATMS environment (a set of axioms from which a formula is derivable) is generalised to include the non-deterministic choices required for the derivation of the formula. We show that this extended ATMS (which we call the D-ATMS), combined with a tableau reasoner extended with a blocking condition to ensure termination, produces correct, complete and minimal explanations for a contradiction in an $\mathcal{ALC}$ ontology where inclusion axioms always contain an atomic concept on the left (which we refer to as dictionary terminologies). We have developed a prototype implementation of our approach which we call AOD[1]. Experimental results comparing AOD,

[1]An Ontology Debugger

MUPSter[82], and the Pellet [86] explanation service are encouraging, and suggest that AOD can outperform MUPSter and Pellet on both synthetic and real-world ontologies.

The rest of chapter is organised as follows. Section 7.2 presents the general framework of AOD. In Section 7.3, we give a high level overview of the reasoner, including the blocking conditions for handling loops. In Section 7.4, we explain the D-ATMS extensions to a 'classical' ATMS in detail. In Section 7.5, we show that the system is correct and complete relative to the reasoner. The experimental results of the prototype and how the system displays explanations are given in Section 7.6 and Section 7.7.

7.2 System Architecture

Our ontology debugging framework, AOD, consists of two components: a tableaux-style reasoner and the D-ATMS, as described in Figure 7.1.

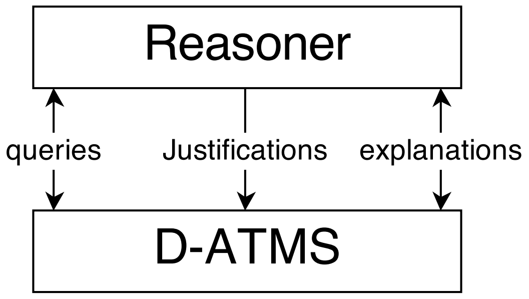

FIGURE 7.1: The components of AOD: a reasoner and the D-ATMS

The reasoner takes as input a set of TBox axioms and a single ABox axiom corresponding to the concept whose emptiness is to be checked, e.g., to check the emptiness a concept A, we add $A(a)$ as an ABox axiom. To check for incoherence, we check whether a contradiction is derivable from the TBox and ABox. The reasoner derives consequences by applying inference rules to axioms and previously derived formulae. Newly derived formulae are communicated to the D-ATMS in the form of inferences. An *inference* $\phi_1, \ldots, \phi_n \overset{r}{\Longrightarrow} \phi$ indicates that the formula ϕ can be derived from the set of formulae $\phi_1, \ldots, \phi_n$ using the inference rule r.

The D-ATMS maintains dependencies between formulae inferred by the reasoner, and computes explanations (minimal reasons) for formulae. To do so, the D-ATMS builds and maintains a justification graph. Each node in the graph corresponds to a formula or a justification (a record of inference; a justification has an outgoing edge to the inferred formula and incoming edges from each of the premises of the inference)[2]. A new inference $\phi_1, \ldots, \phi_n \overset{r}{\Longrightarrow} \phi$ from the reasoner causes the D-ATMS to update the justification graph to record the derivability of ϕ from $\phi_1, \ldots, \phi_n$. If $\phi = \perp$, i.e., if the reasoner has derived an inconsistency, the D-ATMS also records the fact that the antecedents of the justification are known to be inconsistent.

When the reasoner applies an inference rule, it passes the resulting inference to the D-ATMS, causing the D-ATMS to update the justification graph. In addition, the reasoner can query the D-ATMS for the explanations of a previously derived formula ϕ. An *explanation* consists of all minimal sets of axioms from which ϕ can be derived, and, optionally, the sequence of inference rules necessary to derive ϕ from each set of axioms. The explanations returned by the D-ATMS are guaranteed to be correct (in the sense that ϕ is derivable from each of the returned sets of axioms) and minimal (in the sense that ϕ is not derivable from their proper subsets). Explanations are used in AOD in two ways. First, when checking if a constant i is blocked by a constant j in the the $\exists$-rule, the reasoner uses the D-ATMS to determine if the explanations of the concept descriptions which hold for i are a subset of the explanations of the concept descriptions of j. Second, when the reasoner can make no new inferences, the D-ATMS is invoked to compute all explanations for $\perp$.

[2]Note that we use the term justification as it is used in ATMS literature, rather than to mean the minimal set of axioms responsible for an entailment as in, e.g., [10].

7.3 The Reasoner

In this section we introduce the syntax of the logic we are using, and the inference rules.

7.3.1 A Dictionary $\mathcal{ALC}$

The syntax of $\mathcal{ALC}$ includes a set of atomic concepts (unary predicates) $A, A_1, \ldots$, roles (binary predicates) $s, r, \ldots$, and constants $a, b, \ldots$. Complex concepts are built from those using intersection $\sqcap$, union $\sqcup$ (we generalise slightly to n-ary versions of intersection and union), negation $\neg$, existential quantification $\exists s.C$ (which defines the set of objects connected by the role s to an individual defined by concept C), and universal quantification $\forall s.C$ (which defines a set of objects all of whose s-successors are in C). Formulas are formed by stating inclusions between two concepts: $C \sqsubseteq D$, and stating that an individual is described by a concept: $C(a)$. For simplicity, we only allow negation of atomic concepts, since every $\mathcal{ALC}$ formula can be rewritten in negation normal form. We restrict the syntax of inclusions to require that all inclusions have an atomic concept on the left. This is similar to the restriction for unfoldable $\mathcal{ALC}$ terminologies [7, 65], but we do not require in addition that the terminology is acyclic (there may be a chain of inclusions which leads from a concept to itself). We refer to this kind of terminologies as *dictionary* $\mathcal{ALC}$ terminologies. Although a single concept on the left of each inclusion seems a significant restriction, there are quite a few real life ontologies which conform to this restriction, for example, the Biochemistry-primitive ontology from the TONES repository,[3] the

[3] `http://owl.cs.manchester.ac.uk/repository`

Ordnance Survey BuildingsAndPlaces ontology,[4] the Adult Mouse Brain Ontology from the NCBO BioPortal,[5] the Geo ontology [82], the DICE ontology[6], the MGED ontology[7], and a merge of two well-known upper ontologies, a mini-version of SUMO ontology[8] and the CYC ontology[9]. The reasoner is restricted to $\mathcal{ALC}$ rules for the results reported in this work (and some features of the ontologies listed above, such as role inclusions, are ignored), but it is reasonably straightforward to extend the reasoner with additional inference rules.

The reasoner is a tableau reasoner for dictionary $\mathcal{ALC}$ terminologies. It uses essentially the same rules as in [64, 81], together with a blocking condition:

$\sqsubseteq$**-rule** from $A(a)$ and $A \sqsubseteq C$ derive $C(a)$

$\sqcap$**-rule** from $(C_1 \sqcap \ldots \sqcap C_n)(a)$ derive $C_1(a), \ldots, C_n(a)$

$\sqcup$**-rule** from $(C_1 \sqcup \ldots \sqcup C_n)(a)$, derive choices $C_1(a)$, …, $C_n(a)$

$\bot$**-rule** from $A(a)$ and $\neg A(a)$ derive $\bot$

$\forall$**-rule** from $(\forall s.C)(a)$ and $s(a, b)$ derive $C(b)$

$\exists$**-rule** from $(\exists s.C)(a)$ derive $s(a, b), C(b)$ where: b is a new individual, $(\exists s.C)(a)$ has not been used before to generate another new individual.

where A is an atomic concept, C and D are arbitrary concepts, a, b are constants, and s is a role. For ontologies that include disjointness axioms of the form $DJ(A_1, \ldots, A_n)$ stating that the concepts $A_1, \ldots, A_n$ are pairwise disjoint, we add the following inference rule to the reasoner:

[4]`http://www.ordnancesurvey.co.uk/oswebsite/ontology/BuildingsAndPlaces/v1.1/BuildingsAndPlaces.owl`

[5]`http://bioportal.bioontology.org/ontologies/1290`

[6]http://www.mindswap.org/2005/debugging/ontologies/dice.owl

[7]http://www.mged.org

[8]http://www.ontologyportal.org

[9]http://www.opencyc.org

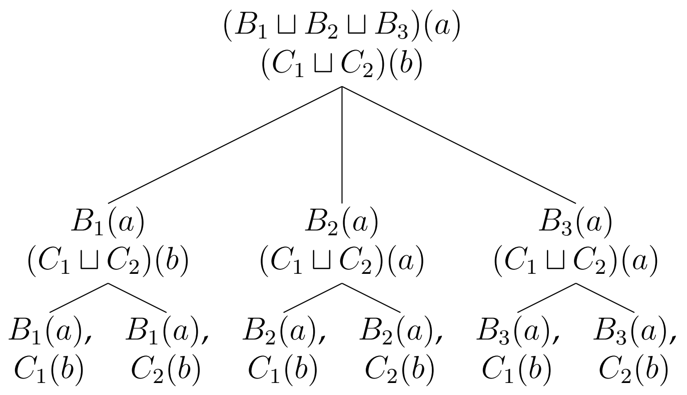

FIGURE 7.2: Tableau with nested disjunctions

dj**-rule** from $DJ(A_1, \ldots, A_n)$ and $A_i(a)$, $A_j(a)$, $i, j \in \{1, \ldots, n\}$, $i \neq j$ derive $\bot$.

The $\sqsubseteq$, $\sqcap$, $\forall$, $\bot$ and dj-rules are straightforward. The $\sqcup$-rule allows us to reason by cases when we encounter a formula of the form $(C_1 \sqcup \ldots \sqcup C_n)(a)$. The $\sqcup$-rule creates branches in the tableaux for each disjunct (choice) $C_1(a)$, ..., $C_n(a)$. A tableau is a tree where nodes are sets of formulae, and children of a node are obtained by applying inference rules to formulae in the node, so that the child node(s) contains all the formulae from the parent node and the newly derived formula. For readability, we will sometimes show only the new formula in a child node, with the understanding that all the formulae higher up on the branch belong to the node as well. If a node contains several disjunctions, for example $B_1 \sqcup B_2 \sqcup B_3(a)$ and $C_1 \sqcup C_2(b)$ as in Figure 7.2, the order in which the disjunction rule is applied does not matter, but once this order is fixed, the choices for the second disjunction are repeated under each of the choices for the first disjunction (see Figure 7.2).

7.3.2 Loops and The Blocking Conditions

The original definition of of unfoldable TBox does not allow cyclic definition of concepts although cyclic definitions are essentially a useful feature for DL modellers. For example, it is straight-forward to define a man who has only male descendants ($Momd$)[10] using a cyclic definition as follows.

$$Momd = Man \sqcap \forall hasChild.Momd$$

It is very hard and nonintuitive for modeller to define such a concept without cyclicity.

However, if we remove one condition of unfoldable TBox (see 3.6) specifying that the right-hand side of a concept definition cannot refer directly or indirectly to the concept name it defines, the following TBox will cause looping in the reasoner.

$$T = \{A(i_1), A \sqsubseteq \exists s.A\}$$

Basically, the $\sqsubseteq$-rules and $\exists$-rule while applying to $A(i_1)$ will create a fresh constant i_2 and an assertion $A(i_2)$, and the same process applies to $A(i_2)$. This will create an infinite chain of assertions of the form:

$$A(i_1) \xrightarrow{\sqsubseteq-rule} \exists s.A(i_1) \xrightarrow{\exists-rule} A(i_2) \xrightarrow{\sqsubseteq-rule} \exists s.A(i_2) \xrightarrow{\exists-rule} \ldots \xrightarrow{\exists-rule} A(i_n) \xrightarrow{\sqsubseteq-rule} \ldots$$

Therefore, it is essential to have a blocking condition for the application of the $\exists$-rule to prevent the generation of similar assertions. However, this condition can be varied depending on the reasoning task. In the following sections, we will look at the blocking condition with and without pinpointing.

[10]This example is given from [6]

7.3.2.1 Blocking Condition without Pinpointing

Recall that in section 7.3.1, the condition of $\exists$-rule is only that there exists an assertion of the form $(\exists s.C)(a)$ and the rule has not been applied on it yet. Because looping might occur, we need another condition to block constant a (or the assertion $(\exists s.C)(a)$) if they cause looping. The reason to block a constant instead of the assertion is that there can be multiple assertions with the same constant causing looping. For example,

$$T_1 = \{A(i_1), A \sqsubseteq \exists s.A \sqcap B, B \sqsubseteq \exists s.B\}$$

can have two assertions with the same constant causing loops $\exists s.A(i_1)$ and $\exists s.B(i_1)$.

Blocking Condition 1

- A constant a_{i+1} is blocked by a constant a_i if for each node in the ATMS whose datum contains a_{i+1}, there exists a node which is similar whose datum contains a_i.
- Two assertions are *similar* if they have the same concept description and different constants (for instance, $A(a_1)$ and $A(a_2)$ are similar while $A(a_1)$) and $B(a_1)$ are not.

This blocking condition will be checked everytime an *exists rule* (in our T_1 ontology, $\exists s.A(i_2)$) is triggered. If i_2 is blocked by another constant (in this case is i_1), the rule will not be fired, and hence the reasoner terminated. For normal reasoning services such as unsatisfiability or consistency checking, this blocking condition is enough because the reasoner only needs to know *whether or not* an assertion is derivable, not *how* it is derived. In the next section, the blocking condition for pinpointing will be examined.

7.3.2.2 Blocking Condition with Pinpointing

A similar loop also occurs in

$T_2 = \{A(i_1), A \sqsubseteq \exists s.A \sqcap B \sqcap C \sqcap D \sqcap E(1), B \sqsubseteq \neg E(2), C \sqsubseteq \forall s.\neg E(3), D \sqsubseteq \forall s.\forall s.\neg E(4)\}$.[11]

The case of T_2 is more complicated than the case of T_1, because if constant i_2 is blocked by constant i_1, $\bot$ is only derivable from $\{1, 2\}$ and $\{1, 3\}$. In fact, $\{1, 4\}$ can also derive $\bot$. It is because node $\neg E(i_1)$ and $\neg E(i_2)$ are considered similar according to *Blocking Condition 1*, while they actually come from different derivations ($\neg E(i_1)$ is derived from $\{\{1, 2\}\}$ while$\neg E(i_2)$ is derived from $\{\{1, 2\}, \{1, 3\}\}$). Therefore, it is also necessary to take into account the environments of nodes in blocking condition to ensure that all explanations are found.

The refined blocking condition is the sames as in *Blocking Condition 1*, but the definition of nodes's similarity should be change to:

Blocking Condition 2

- A constant a_{i+1} is blocked by a constant a_i if for each node in the ATMS whose datum contains a_{i+1}, there exists a node which is similar whose datum contains a_i.
- Two assertions are *similar* iff:
  - they have the same concept description and different constants, and
  - they have the same set of explanations.

This condition is similar to the blocking condition in [9, 59]. In fact, the notion of similarity in *Blocking Condition 2* is similar to $\equiv_{pin}$ in [9]. The difference between this blocking condition and ones in [9, 59] is that this condition applies

[11]This example is adapted from one in [59]

to all nodes in the ATMS (in which each node is an assertion) while the other two apply to two consecutive nodes in a tableaux (in which each node is a set of assertions). With *Blocking Condition 2*, new constants i_3 and i_4 will also be created, and because i_4 is blocked by i_3 according to *Blocking Condition 2*, the reasoner terminates while all explanations for inconsistency can still be found.

Note that the second condition of similarity is too strict, as we only need the explanation set of all assertions of blocked constant to be the subset of the explanation set of all assertions of blocking constant. Therefore, we modify the $\exists$-rule in the beginning of Section 7.3.1 to incorporate the blocking condition as follows:

Blocking Condition 3 Let $assertions(i)$ be the set of all concept descriptions which hold for i. Then a constant i is blocked by a constant j if the following two conditions hold:

- $assertions(i) \subseteq assertions(j)$
- for each $C(i)$ in $assertions(i)$, the set of explanations of the node corresponding to $C(i)$ is a subset of the set of explanations of the node corresponding to $C(j)$.

The $\exists$**-rule** will then become:

$\exists$**-rule** from $(\exists s.C)(a)$ derive $s(a,b), C(b)$ where: b is a new individual, $(\exists s.C)(a)$ has not been used before to generate another new individual, *and a is not blocked*.

It is a standard subset blocking condition for assertions, extended to ensure that assertions of the blocked constant do not have any new and different ways of being derivable (compared to the assertions of the blocking constant), which may result in new explanations being produced.

The reasoner derives consequences by applying inference rules to axioms and previously derived formulae. In order to deal with cyclic terminologies, the reasoner operates in phases. In the *odd phase*, all inference rules apart from the instances of $\exists$-rule are applied. When no inference rules other than the $\exists$-rule are applicable, execution switches to the *even phase*, in which non-blocked instances of the $\exists$-rule are applied.

Each possible inference is generated exactly once. Two inferences are the same if they have the same antecedents and result from the application of the same inference rule. This means that the reasoner will generate multiple inferences with the same formula as the consequent, if the formula can be derived from different antecedents or different inference rules. Newly derived formulae may form the antecedents of further inferences, and the cycle repeats until no new inferences can be made.[12]

In order to reduce the time required for pattern matching and the search for derivations of a contradication, we pre-process the ontology depending on the input ABox, essentially computing the concepts reachable from the ABox and the corresponding inclusion axioms, or the logical module for the Abox (see, for example, [41, 51]). Since the input ontologies always have an atomic concept on the left of the inclusion axioms, there exists a quite straightforward algorithm which guarantees completeness of the resulting module.

7.4 The D-ATMS

Like the original ATMS [23], the D-ATMS maintains dependencies between formulae inferred by the reasoner and computes minimal sets of axioms from

[12]In particular, the reasoner does not stop after a contradiction is derived on a branch, but continues to apply inference rules until no new rule applications are possible.

which a formula is derivable.[13] In this section, we explain how the ATMS introduced in [23] is generalised to deal with disjunctions.

7.4.1 Dealing with Disjunctions

The original ATMS [23] can only represent Horn-clause inferences supplied by the reasoner such as $A_1 \cap A_2 \cap A_3 \to B$. This is fine with the less expressive description logics such as $\mathcal{EL}$, as there is no disjunctive concept description. However, it is not the case for unfoldable $\mathcal{ALC}$, as now disjunction is allowed under the $\sqcup$-rule in Section 7.3.1:

$\sqcup$-rule from $(C_1 \sqcup \ldots \sqcup C_n)(a)$, derive choices $C_1(a), \ldots, C_n(a)$

Recall from Section 3.2.4 that a tableau is a tree where each node is a set of assertions, the way tableaux-like reasoners deal with disjunction is quite intuitive. Everytime a $\sqcup$-rule is applied to a node of the tableau, the node is divided into two child-nodes, and each of them represents the disjunctive branch of the parent node (see Figure 7.3). Then the reasoning rules will apply to the children nodes instead of the parent node.

$$\begin{tikzcd} & B(a),\ (C_1 \sqcup C_2)(a) \arrow[dl, no head] \arrow[dr, no head] & \\ B(a),\ (C_1 \sqcup C_2)(a) \\ C_1(a) & & B(a),\ (C_1 \sqcup C_2)(a) \\ C_2(a) \end{tikzcd}$$

FIGURE 7.3: Tableau reasoning with disjunctions

In an ATMS, a node records only one assertion, and hence it will be harder to record an inference resulted from the $\sqcup$-rule. For example, in Figure 7.4, how

[13]An ATMS typically also provides additional functionality, e.g., interpretation construction; as these capabilities are not required for our approach we do not discuss them here.

can one distinguish between an inference from conjunction and another inference from disjunction? Therefore, there should be a way for derived nodes such as $C_1(a)$ and $C_2(a)$ to recognise whether they or their ancestor nodes are derived from disjunctive inferences. To do so, we need to embed the information of all disjunctive choices which are used to derive a node in its label, in addition to the set of nodes where it is derived from as in the original ATMS.

$$\begin{tikzcd}
& (C_1 \sqcap C_2)(a) \arrow[dl, no head] \arrow[dr, no head] & & & (C_1 \sqcup C_2)(a) \arrow[dl, no head] \arrow[dr, no head] & \\
C_1(a) & & C_2(a) & C_1(a) & & C_2(a)
\end{tikzcd}$$

FIGURE 7.4: The inference from a conjunction and a disjunction might be recored by a typical ATMS

A disjunctive choice has two parts to record: the disjunction and a choice (i.e., a disjunct). As the result, instead of recording only a set of axioms as in the original ATMS, to be able to record disjunctive inferences, an environment has to be extended to include a sequence of disjunctive choices from which, the set of axioms can consistently derive a node. To sum up, there are three data-structures in the original ATMS which can be added (or extended) to allow reasoning with non-Horn clause formulae: *a*) another kind of justification to record disjunctive inference supplied by the reasoner; *b*) a disjunctive choice, which includes a unique disjunction and one disjunct; and *c*) an extended environment of a node, which now needs to include the sequence of disjunctive choices have been made to derive that node. In the following section, we will look in more details at how the original ATMS is extended to deal with disjunctions.

7.4.2 The D-ATMS Data-structures: Nodes, Justifications, Environments and Labels

The D-ATMS maps reasoner inferences to an internal representation based on nodes and justifications.[14] Each formula is represented by a D-ATMS *node*. We denote the node corresponding to a formula ϕ by n_ϕ. Axioms are represented by *axiom nodes*, and inconsistency is represented by a distinguished *false node*, $n_\perp$. In the interests of readability, we will often refer to a formula node n_ϕ by the formula ϕ it represents. To compute derivability, the D-ATMS builds a justification graph. *Justification nodes* record the fact that a node (the consequent) can be derived from a set of other nodes (the antecedents): they have an outgoing edge to the consequent and incoming edges from each of the antecedents. A node may be the consequent of more than one justification (recording the different ways in which it can be derived), and be an antecedent in other justifications (recording the inferences that can be made using it).

Definition 7.1. A justification is a structure $j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi$, where $n_{\phi_1}, \ldots, n_{\phi_k}$ are nodes corresponding to the antecedents of an inference, n_ϕ is a node corresponding to the consequent of the inference, and j is the justification id, a unique, sequentially assigned integer that identifies the justification.

As ids are unique, we will often refer to a justification by its id. The D-ATMS distinguishes two different types of justification: deterministic justifications and non-deterministic justifications. Non-deterministic justifications are produced by the $\sqcup$-rule and have a choice (a formula appearing as a disjunct in a disjunction) as the consequent and a single antecedent consisting of the disjunction in which the choice appears. Non deterministic justifications are of the form $j : n_d \Rightarrow n_{\psi_i}$, where d is of the form $\psi_1 \sqcup \ldots \sqcup \psi_k$, $1 \leq i \leq k$. Deterministic justifications may be derived using any of the decomposition rules except the

[14]Note that we use the term justification as it is used in ATMS literature, rather than to mean the minimal set of axioms responsible for an entailment as in, e.g., [10].

$\sqcup$-rule, and may have any formulae except a disjunction as antecedents. Deterministic justifications are of the form $j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi$.

When reasoning begins, the D-ATMS contains only axiom nodes. As the reasoner derives consequences, it sends the inferences to the D-ATMS. A justification is added linking the nodes representing the antecedents of the inference to the node representing the consequent (if no node exists for the derived formula, one is created). The reasoner may designate certain sets of formulae as inconsistent (nogood in ATMS terminology) by providing a justification for $n_\perp$.

Each node in the justification graph has a *label* consisting of a set of environments.

Definition 7.2 (environment)**.** An environment e is a pair $(\mathcal{A}, \mathcal{C})$ where $\mathcal{A}$ is a set of axioms and $\mathcal{C}$ is a sequence of choice sets $[c_1, \ldots, c_k]$ of length $k \geq 0$. Each choice set c_i is a pair (d_i, b_i) where $d_i = \psi_1 \sqcup \ldots \sqcup \psi_n$ is a disjunction and $b_i \subset \{\psi_1, \ldots, \psi_n\}$ is a set of choices for d_i (i.e., a subset of the disjuncts appearing in the disjunction).

An environment represents a set of axioms and choices under which a particular formula holds.[15] The presence of an environment $(\mathcal{A}, \mathcal{C})$ in the label of a node n_ϕ indicates that ϕ can be derived from the axioms $\mathcal{A}$ together with a sequence of choices from $\mathcal{C}$. The choice sequence corresponds to a (set of) tableau branch(es): each *choice* consists of a disjunction d_i and one or more of the disjuncts appearing in d_i. If ϕ can be derived from all the disjuncts appearing in d_i, we have eliminated dependency on all choices for d_i, and the choice set for d_i can be removed from $\mathcal{C}$. If the sequence of choice sets is empty, then ϕ does not depend on any choices (i.e., it can be derived from only the axioms $\mathcal{A}$). For example, the presence of the environment $(\{\phi_1, \ldots, \phi_k\}, [\,])$ in the label of a node n_ϕ means that ϕ has been derived by the reasoner from the axioms $\phi_1, \ldots, \phi_k$.

[15] In a 'classical' ATMS [23], environments do not contain choice sequences.

Environments in the D-ATMS thus capture the branching structure of a tableau. For example, in the tableau in Figure 7.2 an environment for $B_1(a)$ will have a choice sequence $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a))]$ and $C_1(b)$ will have a choice sequence $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a)),\ ((C_1 \sqcup C_2)(b), C_1(b))]$. The order of choice sets in a choice sequence comes from the order in which the $\sqcup$-rule is applied to disjunctions on the corresponding branch. If one choice sequence corresponds to a prefix of another, then the first choice sequence depends on fewer disjunctive choices. This intuition may be helpful when considering the definition of subsumption for environments below.

The *label* of a node contains the set of environments from which the formula corresponding to the node can be derived. The label of $n_\perp$ consists of a set of inconsistent environments or nogoods. Initially, the labels of all nodes in the justification graph other than axiom nodes are empty, and the label of each axiom node contains a single environment consisting of the axiom itself and an empty sequence of choice sets.

7.4.3 Example

As an example, consider the following TBox inspired by the MadCow example from the OilEd tutorial:

ax1 $Sheep \sqsubseteq Animal$

ax2 $Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal$

ax3 $MadCow \sqsubseteq Cow \sqcap \exists eats.(Sheep \sqcup Cow)$

we also add the assumption $MadCow(a)$. The inferences made by the reasoner give rise to the following justifications (note that $Animal(b)$ has two justifications):

j_1 $MadCow(a), MadCow \sqsubseteq Cow \sqcap \exists eats.(Sheep \sqcup Cow) \Rightarrow Cow \sqcap \exists eats.(Sheep \sqcup Cow)(a)$

j_2 $Cow \sqcap \exists eats.(Sheep \sqcup Cow)(a) \Rightarrow Cow(a)$

j_3 $Cow \sqcap \exists eats.(Sheep \sqcup Cow)(a) \Rightarrow \exists eats.(Sheep \sqcup Cow)(a)$

j_4 $Cow(a), Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal, \Rightarrow Animal \sqcap \forall eats.\neg Animal(a)$

j_5 $\exists eats.(Sheep \sqcup Cow)(a) \Rightarrow eats(a, b)$

j_6 $\exists eats.(Sheep \sqcup Cow)(a) \Rightarrow (Sheep \sqcup Cow)(a)$

j_7 $(Animal \sqcap \forall eats.\neg Animal)(a) \Rightarrow Animal(a)$

j_8 $(Animal \sqcap \forall eats.\neg Animal)(a) \Rightarrow \forall eats.\neg Animal(a)$

j_9 $eats(a, b), \forall eats.\neg Animal(a) \Rightarrow \neg Animal(b)$

j_{10} $(Sheep \sqcup Cow)(a) \Rightarrow Sheep(a)$ (non-deterministic)

j_{11} $(Sheep \sqcup Cow)(a) \Rightarrow Cow(a)$ (non-deterministic)

j_{12} $Sheep(b), Sheep \sqsubseteq Animal \Rightarrow Animal(b)$

j_{13} $Animal(b), \neg Animal(b) \Rightarrow \perp$

j_{14} $Cow(b), Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal \Rightarrow (Animal \sqcap \forall eats.\neg Animal)(b)$

j_{15} $(Animal \sqcap \forall eats.\neg Animal)(b) \Rightarrow Animal(b)$

j_{16} $(Animal \sqcap \forall eats.\neg Animal)(b) \Rightarrow \forall eats.\neg Animal(b)$

and the justification graph is shown in Figure 7.5. Each node in the justification graph is labelled with a set of environments: the minimal sets of axioms from which the corresponding formula is derivable (i.e., an explanation). For example, $Animal(a)$ in Figure 7.5 would have an environment $\{Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal, MadCow \sqsubseteq Cow \sqcap \exists eats.(Sheep \sqcup Cow), MadCow(a)\}$.

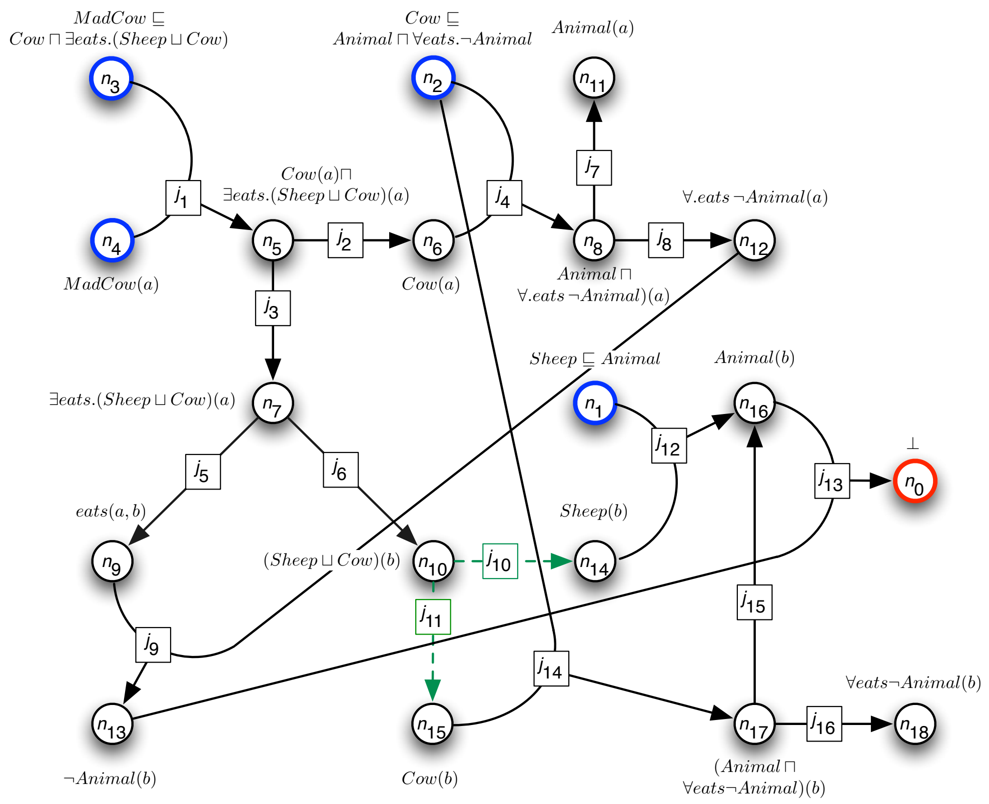
$MadCow \sqsubseteq Cow \sqcap \exists eats.(Sheep \sqcup Cow)$
$Cow \sqsubseteq Animal \sqcap \forall eats.\neg Animal$
$Animal(a)$
n_3
n_2
n_{11}
j_1
$Cow(a) \sqcap \exists eats.(Sheep \sqcup Cow)(a)$
j_4
j_7
$\forall.eats\,\neg Animal(a)$
n_4
n_5
j_2
n_6
n_8
j_8
n_{12}
$MadCow(a)$
$Cow(a)$
$Animal \sqcap \forall.eats\,\neg Animal)(a)$
j_3
$Sheep \sqsubseteq Animal$
$Animal(b)$
$\exists eats.(Sheep \sqcup Cow)(a)$
n_7
n_1
n_{16}
$\perp$
j_5
j_6
j_{12}
j_{13}
n_0
$eats(a,b)$
$Sheep(b)$
n_9
$(Sheep \sqcup Cow)(b)$
n_{10}
j_{10}
n_{14}
j_{11}
j_{15}
j_9
j_{14}
$\forall eats\neg Animal(b)$
n_{13}
n_{15}
n_{17}
j_{16}
n_{18}
$\neg Animal(b)$
$Cow(b)$
$(Animal \sqcap \forall eats\neg Animal)(b)$

FIGURE 7.5: Justification graph. Formula nodes are round, axioms are blue, $\perp$ is red. Justification nodes are square, non-deterministic justifications are green with dashed arrows.

7.4.4 Label Computation

We now describe how labels are computed from the justifications generated by the reasoner.

To define the D-ATMS algorithms for computing labels, we need the following primitive operations on environments and labels which generalise and extend the corresponding notions in [23].

We say that a choice sequence $\mathcal{C}_1$ is a prefix of a choice sequence $\mathcal{C}_2$, $\mathcal{C}_1 \preceq \mathcal{C}_2$, if $\mathcal{C}_1 = [(d_1, b_1), \ldots, (d_k, b_k)]$ and $\mathcal{C}_2 = [(d'_1, b'_1), \ldots, (d'_n, b'_n)]$, $k \leq n$ and for every $i \leq k$, $d_i = d'_i$ and $b'_i \subseteq b_i$. $\mathcal{C}_1 \prec \mathcal{C}_2$ iff $\mathcal{C}_1 \preceq \mathcal{C}_2$ and $\mathcal{C}_2 \not\preceq \mathcal{C}_1$.

Definition 7.3 (Subsumption of environments). An environment $(\mathcal{A}_1, \mathcal{C}_1)$ subsumes an environment $(\mathcal{A}_2, \mathcal{C}_2)$, $(\mathcal{A}_1, \mathcal{C}_1) \subseteq_s (\mathcal{A}_2, \mathcal{C}_2)$ iff $\mathcal{A}_1 \subseteq \mathcal{A}_2$, and $\mathcal{C}_1 \preceq \mathcal{C}_2$.

$(\mathcal{A}_1, \mathcal{C}_1) \subset_s (\mathcal{A}_2, \mathcal{C}_2)$ iff $(\mathcal{A}_1, \mathcal{C}_1) \subseteq_s (\mathcal{A}_2, \mathcal{C}_2)$ and $(\mathcal{A}_2, \mathcal{C}_2) \not\subseteq_s (\mathcal{A}_1, \mathcal{C}_1)$.

An environment e is *nogood* if it is subsumed by an environment in the label of the false node $n_\perp$. Note that for sequences of binary disjunctions, the condition for subsumption of environments above becomes $b'_i = b_i$, and can be rephrased more simply as $\mathcal{C}_1$ is a prefix of $\mathcal{C}_2$. For *n-ary* disjunctions it is possible to have $b'_i \subset b_i$. For example, $(A, [((B_1 \sqcup B_2 \sqcup B_3)(a), \{B_1(a), B_2(a)\})])$ subsumes (is more informative than) $(A, [((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a))])$ because the latter depends on a more specific set of choices.

Definition 7.4 (Union of environments)**.** The union of two environments $e_1 = (\mathcal{A}_1, \mathcal{C}_1)$ and $e_2 = (\mathcal{A}_2, \mathcal{C}_2)$, $e_1 \cup_\leq e_2 = (\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup_\leq \mathcal{C}_2)$ if $\mathcal{C}_1$ and $\mathcal{C}_2$ are sequences of choice sets for which $\mathcal{C}_1 \cup_\leq \mathcal{C}_2$ is defined, otherwise $e_1 \cup_\leq e_2$ is not defined. $\cup_\leq$ for sequences of choice sets is defined as follows:

1. if $\mathcal{C}_1 \preceq \mathcal{C}_2$ then $\mathcal{C}_1 \cup_\leq \mathcal{C}_2 = \mathcal{C}_2$;
2. if $\mathcal{C}_2 \preceq \mathcal{C}_1$ then $\mathcal{C}_1 \cup_\leq \mathcal{C}_2 = \mathcal{C}_1$;
3. for all other cases, $\mathcal{C}_1 \cup_\leq \mathcal{C}_2$ is not defined.

Intuitively, environments of two antecedents can be combined by $\cup_\leq$ to form an environment of the consequent if the antecedents belong to the same branch of the tableau. The consequent belongs to the lower of the two disjunctive nodes in the tableau to which the antecedents belong.

Definition 7.5 (Merge of environments)**.** The merge of two environments $e_1 = (\mathcal{A}_1, \mathcal{C}_1)$ and $e_2 = (\mathcal{A}_2, \mathcal{C}_2)$, $e_1 \cup_+ e_2 = (\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup_+ \mathcal{C}_2)$ if $\mathcal{C}_1$ and $\mathcal{C}_2$ are sequences of choice sets for which $\cup_+$ is defined. Otherwise, $e_1 \cup_+ e_2$ is not defined. $\cup_+$ for sequences of choice sets is defined as follows:

1. if $\mathcal{C}_1 = [(d_1, b_1), \dots, (d_n, b_n)]$ and $\mathcal{C}_2 = [(d'_1, b'_1), \dots, (d'_n, b'_n)]$, $n \geq 1$, and for every $i < n$ $d_i = d'_i$ and $b'_i = b_i$ (in other words, $\mathcal{C}_1$ and $\mathcal{C}_2$ are the same apart from their last element), $d_n = d'_n$, $b_n \neq b'_n$, then

   (a) if $b_n \cup b'_n$ does not include all the disjuncts in d_n, then $\mathcal{C}_1 \cup_+ \mathcal{C}_2 = [(d_1, b_1), \ldots, (d_n, b_n \cup b'_n)]$

   (b) $\mathcal{C}_1 \cup_+ \mathcal{C}_2 = [(d_1, b_1), \ldots, (d_{n-1}, b_{n-1})]$ otherwise;

2. for all other cases, $\mathcal{C}_1 \cup_+ \mathcal{C}_2$ is not defined.

Intuitively, if the same formula belongs to all children of a disjunctive node in a tableau, then it can be lifted 'up' to the parent, otherwise, $\cup_+$ merges two subtrees into one subtree where the formula belongs to all children. Recall that the label of a node is the set of all environments from which the node can be derived.

Definition 7.6 (Union of labels)**.** The union of two labels L_1 and L_2, $L_1 \cup_+ L_2 = L_1 \cup L_2 \cup \{e_1 \cup_+ e_2 \mid e_1, e_2 \in L_1 \cup L_2\}$.

7.4.5 Lazy Label Update Propagation

We can now explain how the standard ATMS label computation algorithms are extended to handle disjunctions. This section briefly presents a summary of the lazy approach to label computation and the algorithms to compute correct labels.

Recall that in the original ATMS [23], labels must be correct globally, i.e., the label of every node in the dependency graph has to be computed to maintain global correctness and completeness. However, this property might be unnecessary in some cases. For example, for axiom pinpointing and ontology debugging context, we are only interested in the derivation of $\perp$; therefore, all nodes in the dependency graph which are not involved in the derivation of $\perp$ become irrelevant, and hence their labels do not need to be computed or updated. In other words, we only propagate label updates when needed. To do so, we need to define a set of target nodes $N = \{n_{\phi_1}, \ldots, n_{\phi_m}\}$ whose labels (i.e., explanations) have to be computed and completed. For example, when checking if an inference is blocked (see Section 7.3), the target nodes are nodes corresponding to the assertions about a constant. When computing explanations for $\perp$, the target node is $n_\perp$.

Restricting label computation in the D-ATMS to a particular set of nodes (in the evaluation of the blocking condition and to $n_\perp$ in explanation generation) has some similarities with lazy label evaluation in assumption-based truth maintenance systems, e.g., [58], and to work on focusing the ATMS, e.g., [31, 32, 37]. Such approaches have been shown to offer significant performance improvements relative to the ATMS described in [23].

Firstly the union of the justification closure $J = J_{n_{\phi_1}} \cup \ldots \cup J_{n_{\phi_m}}$ for each node $n_{\phi_i} \in N$ is computed. Each $J_{n_{\phi_i}} \subseteq J$ is the set of justifications that have n_{ϕ_i} as a consequent, together with the justifications of the antecedents of those justifications, and so on until we reach justifications whose antecedents are axiom nodes (assumptions) or which are already included in $J_{n_{\phi_1}} \subseteq J$.[16] For example, when computing explanations for $\perp$ in the MadCow example in Figure 7.5, the relevant part is the graph without the justifications j_7, j_{16} and the nodes n_{11}, n_{18}.

The justifications in J are processed in order of justification id (recall that a smaller justification id means earlier inference). For each justification

$$j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi \in J,$$

we first compute a *label update* — a set of new environments computed from the Cartesian product of the labels of the antecedents to be merged into the label of n_ϕ. There are two cases: if j is a nondeterministic justification, this is done

[16]For reasons of efficiency, we exclude from J any justification used in label computation at a previous cycle of the reasoner, as the D-ATMS labels have already been updated with these justifications.

by the procedure UPDATE-NONDET-JUSTIF, otherwise, if j deterministic, by the procedure UPDATE-DET-JUSTIF. We then merge the label update into the label of the consequent. If $\phi = \bot$ (i.e., if j is a justification for $n_\bot$) this is done by the procedure UPDATE-FALSE-LABEL, and by UPDATE-NODE-LABEL otherwise. If $\phi \neq \bot$ and applying the label update results in a change in the label of n_ϕ, we propagate the new label to nodes reachable by following already processed (having a smaller id) justification links $j' \in J, j' < j$ (since we discovered a new way to derive the formulae which are their consequents). The process terminates when the labels of all reachable nodes have been updated. In the worst case all justifications in the justification graph must be traversed, but the process is guaranteed to terminate.

Below we give algorithms for each step of the label computation process. The algorithms are similar to those in [25], but have been extended to handle non-deterministic justifications.

Algorithm 7.1 Update a non-deterministic justification

```
procedure UPDATE-NONDET-JUSTIF($j : n_d \Rightarrow n_{\psi_i}, m, J$)
    $L \leftarrow \{(\mathcal{A}, \mathcal{C}' + (d, \psi_i)) \mid (\mathcal{A}, \mathcal{C}) \in label(n_d) \wedge$
            $\exists n'(\mathcal{A}', \mathcal{C}') \in label(n') \wedge \mathcal{C} \preceq \mathcal{C}' \wedge$
           $\neg\exists n''(\mathcal{A}'', \mathcal{C}'') \in label(n'') \wedge \mathcal{C}' \prec \mathcal{C}''\}$
    UPDATE-NODE-LABEL($n_{\psi_i}, L, m, J$)
end procedure
```

The first procedure, UPDATE-NONDET-JUSTIF takes a non-deterministic justification $j : n_d \Rightarrow n_{\psi_i}$, the justification closure of the target nodes J, and m, the id of the justification in J currently being processed as arguments, and computes a *label update* L for the label of the consequent choice, n_{ψ_i}. The axioms appearing in the environments in L are the same as the axioms in the environments of the label of n_d, as ψ_i is derivable from the same axioms as d. However the sequences of choice sets must be updated to record the dependency on the choice ψ_i. As explained above, a sequence of choice sets encodes the branches in the tableau on which the formula ψ_i has been derived, and new branches must be added under each existing branch in the tableau where the disjunction is derivable. To reflect this tableau structure in the label of n_{ψ_i}, for each environment $(\mathcal{A}, \mathcal{C})$ appearing in the label of n_d we compute the set of choice sequences of maximal length appearing in any label $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$ where $\mathcal{C}$ is a prefix of $\mathcal{C}_s$, $1 \le s \le k$, and add an environment $(\mathcal{A}, \mathcal{C}_s + (d, \{\psi_i\}))$ to the label update L. For example, in Figure 7.2, when the $\sqcup$-rule is applied to $(C_1 \sqcup C_2)(b)$, the only choice sequence appearing in its label is $[\,]$. The set of choice sequences of maximal length which have $[\,]$ as a prefix are $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a))]$, $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_2(a))]$, and $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_3(a))]$. The choice sequences in the label update of $C_1(b)$ and $C_2(b)$ are therefore $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a)), ((C_1 \sqcup C_2)(a), C_1(a))]$, $[((B_1 \sqcup B_2 \sqcup B_3)(a), B_1(a)), ((C_1 \sqcup C_2)(a), C_2(a))]$, etc. We then call UPDATE-NODE-LABEL to update the label of the consequent node n_{ψ_i} with L.[17] (Note that m and J are not used by UPDATE-NONDET-JUSTIF and are simply passed through to UPDATE-NODE-LABEL.)

Algorithm 7.2 Update a deterministic justification

```
procedure UPDATE-DET-JUSTIF($j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi$, $m$, $J$)
    if $j \in J \wedge j < m$ then
        $L \leftarrow \{e_1 \cup_\le \ldots \cup_\le e_k \mid e_i \in label(n_{\phi_i}), 1 \le i \le k\}$
        $L \leftarrow \{e \mid e \in L, \neg\exists e' \in L \wedge e' \subset_s e\}$
        $L \leftarrow \{e \mid e \in L, \neg\exists e' \in label(n_\perp) \wedge e' \subseteq_s e\}$
        if $n_\phi = n_\perp$ then
            UPDATE-FALSE-LABEL($n_\perp, L$)
        else
            UPDATE-NODE-LABEL($n_\phi, L, m, J$)
        end if
    end if
end procedure
```

The corresponding procedure for deterministic justifications, UPDATE-DET-JUSTIF, takes a deterministic justification $j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi$, the justification closure of the target nodes J, and m, the id of the justification in J currently being processed as arguments. If $j : n_{\phi_1}, \ldots, n_{\phi_k} \Rightarrow n_\phi$ is in the justification closure of the

[17]Note that, without loss of generality, we assume that $\perp$ does not appear as a disjunct in a disjunction.

target nodes and the justification id j is less than the id of the justification in J currently being processed m, UPDATE-DET-JUSTIF computes a label update L for the label of the consequent n_ϕ. For every k-tuple of environments from the labels of $n_{\phi_1}, \ldots, n_{\phi_k}$ (every way to derive the premises) we take their $\cup_\leq$ union (which means, we only combine derivations on the same branch), remove any of the resulting environments which are subsumed (to guarantee minimality), and then remove nogoods. If j is a justification for $n_\perp$, UPDATE-DET-JUSTIF calls UPDATE-FALSE-LABEL to record that the antecedents of j are nogood (i.e., inconsistent). Otherwise it calls UPDATE-NODE-LABEL to update the label of the consequent node n_ϕ with the label update L.

Algorithm 7.3 Update the label of a node and propagate to consequents

```
procedure UPDATE-NODE-LABEL($n_\phi, L, m, J$)
    $L' \leftarrow label(n_\phi) \cup_+ L$
    $L' \leftarrow \{e \mid e \in L', \neg\exists e' \in L' \wedge e' \subset_s e\}$
    $L' \leftarrow \{e \mid e \in L', \neg\exists e' \in label(n_\perp) \wedge e' \subseteq_s e\}$
    if $label(n_\phi) \neq L'$ then
        $label(n_\phi) \leftarrow L'$
        for all justifications $j : n_\phi \in antecedents(j)$ do
            UPDATE-DET-JUSTIF($j, m, J$)
        end for
    end if
end procedure
```

The procedure UPDATE-NODE-LABEL takes a (consequent) node n_ϕ, a label update L, the justification closure of the target nodes J, and the id of the justification in J currently being processed m, as arguments, and updates the label of n_ϕ with L. In doing so it ensures that the new label for n_ϕ is minimal and consistent. If the update results in a change in the label of n_ϕ, the new label is propagated through all justifications j where n_ϕ is an antecedent, by calling

UPDATE-DET-JUSTIF with j, m and J as arguments. (Note that, as with UPDATE-NONDET-JUSTIF, m and J are not used by UPDATE-NODE-LABEL and are simply passed through to UPDATE-NONDET-JUSTIF.)

Algorithm 7.4 Remove nogood environments from node labels

```
procedure UPDATE-FALSE-LABEL($n_\perp, L$)
    $L' \leftarrow label(n_\perp) \cup_+ L$
    $L' \leftarrow \{e \mid e \in L', \neg\exists e' \in L' \wedge e' \subset_s e\}$
    if $label(n_\perp) \neq L'$ then
        $label(n_\perp) \leftarrow L'$
        for all nodes $n$ in the justification graph do
            $label(n) \leftarrow \{e \mid e \in label(n), \neg\exists e' \in label(n_\perp) \wedge e' \subseteq_s e\}$
        end for
    end if
end procedure
```

The procedure UPDATE-FALSE-LABEL takes the distinguished node $n_\perp$ and a label update L as arguments, and updates the label of $n_\perp$ with L. In doing so, it ensures that the label of $n_\perp$ is minimal. If the update results in a change in the label of $n_\perp$, any environment subsumed by a new nogood in L is removed from the labels of all nodes, ensuring that all labels (other than that of $n_\perp$) are consistent.

7.5 Correctness

Here we sketch the proofs that, given an inconsistent set of formulae Γ (consisting of a dictionary $\mathcal{ALC}$ TBox and some concept instances), the D-ATMS will return all minimal explanations for the inconsistency. More precisely, it will return as the label of $n_\perp$ the set of all $\Gamma' \subseteq \Gamma$ such that (i) $\Gamma' \models \perp$ and (ii) there is no $\Gamma'' \subset \Gamma'$ such that $\Gamma'' \models \perp$.

In order to show this, we need to establish first that the reasoner has a sound and complete set of decomposition rules and is guaranteed to terminate, and second that the D-ATMS keeps a correct record of the inferences performed by the reasoner. That is, for every chain of rule applications which derives a formula from Γ, there is a corresponding environment in the label of the formula, that all environments correspond to such derivations, and that all environments are minimal. Also, that the D-ATMS maintains the tree structure of the corresponding semantic tableau in the form of choice sequences in environments and when it receives a non-deterministic justification from the reasoner, it creates the required number of branches at the right level in the tree.

The first set of theorems applies to the correctness of the reasoner.

Theorem 7.7 (Reasoner soundness)**.** *The reasoner's rules are sound: if given a set Γ it derives $\perp$, then Γ is unsatisfiable.*

Proof. The set of rules the reasoner uses (if the $\sqcup$ rule is interpreted as a branching rule) are standard rules for $\mathcal{ALC}$, see, for example [8]. □

Theorem 7.8 (Reasoner refutation completeness)**.** *Given an unsatisfiable set Γ, the reasoner derives $\perp$ and finds all possible derivations of $\perp$.*

Proof. If the set Γ is unsatisfiable, the reasoner will find a derivation of $\perp$ on all branches (this is again standard and follows from the refutation completeness of the tableau rules used by the reasoner plus the subset blocking condition: see for example [8, 18]). Since the D-ATMS reasoner does not terminate until there are no applicable rules, it will find *all* derivations of $\perp$ from Γ (since it is refutation-complete, it will find a derivation of $\bot$ from every inconsistent subset Γ' of Γ). Note that the blocking condition ensures that there are no different derivations of $\bot$ which could have been found by using formulae containing (descendants of) the blocked constant, since every such derivation can be simulated using the blocking constant (which has all the concept descriptions with the same explanations as the blocked constant).

□

Theorem 7.9. *The reasoner terminates.*

Proof. The blocking condition ensures that only finitely many new constants will be introduced (see for example [18]; our blocking condition requires in addition that the explanations for each property of the blocked constant are not new, but there are finitely many possible environments, hence there are finitely many possible explanations). This in turn means that only finitely many different formulae will be derived, and each formula can be used in finitely many ways as a premise for an inference. This means that although the reasoner does not terminate immediately when a contradiction discovered (as is customary with tableaux reasoners), it will still produce only a finite number of different rule instances, until its termination condition (no new inferences) is met. □

We now turn to the properties of the D-ATMS proper. Because only the nodes which are the consequent of a justification in the justification closure have their labels computed, the following theorems only apply to those nodes and the formulae corresponding to them.

The following theorem states that the D-ATMS only records environments for a formula that correspond to a valid derivation of the formula by the reasoner.

Theorem 7.10 (Soundness). *For each node n_ϕ and each environment in its label, there is a sequence of tableau rule applications produced by the reasoner, such that the only* *axioms used in the derivation are in the non-disjunctive part of the environment, and the disjunctive part encodes the subtree where ϕ has been derived.*

Proof. The proof is by induction on the longest chain of justifications connecting the formula to the axioms in the environment. Environments are added in response to receiving an inference from the reasoner which in turn corresponds to an inference rule application. Depending on the type of the justification this is handled by Algorithm 1 (non-deterministic justifications, or $\sqcup$-rule applications) or Algorithm 2 (deterministic justifications, applications of all other rules). Algorithm 1 maintains the branching structure of the $\sqcup$-rule applications corresponding to that of a standard tableaux reasoner.

If the antecedents of the justification are axioms, the theorem follows immediately because we have a one step derivation of the formula from those axioms.

The inductive step is routine, and corresponds to adding another inference step to a valid derivation.

After the environment which correctly reflects the axioms used in the derivation is added to the label of the node corresponding to the formula, it may be updated in several ways. First of all, if the environment is discovered to be a superset of another environment or to be inconsistent, it may be removed; this does not violate soundness. Second, it may be updated if it is discovered that the formula is now derivable on all branches of some disjunction; this also corresponds to a valid inference step. □

The following theorem states that for every possible way of deriving a formula from Γ, the D-ATMS records the set of axioms Γ' used in the derivation as an environment in the label of the node corresponding to the formula.

Theorem 7.11 (Completeness relative to reasoner). *Every set of axioms Γ' from which ϕ can be derived given the set of justifications produced by the reasoner, is a* *superset of the axioms appearing in some environment in the label of the corresponding node n_ϕ.*

Proof. By induction on the length of the derivation (the length of the chain of justification produced by the reasoner). For a one step derivation (where the corresponding justification for n_ϕ has axioms as antecedents) this is immediate. For a k step derivation, assume that the justification for n_ϕ has n_{ϕ_1} and n_{ϕ_2} as antecedents, and that all known derivations for ϕ_1 use one of the sets of axioms $e_1, \ldots, e_m$ and for ϕ_2, $e'_1, \ldots, e'_{m'}$. Since ϕ_1 and ϕ_2 occur as the k-1st and k-2nd steps in the derivation of ϕ, the inductive hypothesis applies. Hence all ways to derive ϕ should use $e_1 \cup_\leq e'_1$ or $e_1 \cup_\leq e'_2$, ..., or $e_m \cup_\leq e'_{m'}$. This set of environments will be added to the label of n_ϕ by Algorithms 1 and 2. The 'superset' comes from the fact that Algorithms 2 and 3 check for subsumption of environments and remove the ones which contain redundant axioms. □

This theorem together with Theorem 7.8 ensures completeness of AOD: all possible explanations for $\perp$ will be returned as the label of $n_\perp$.

The next property we need is that the environments are minimal. After the reasoner terminates having produced all possible derivations of $\perp$, all axioms in each environment are guaranteed to be essential for the derivation.

Theorem 7.12 (Minimality)**.** *No environment in any node's label is a subset of any other.*

Proof. Guaranteed by subsumption tests in Algorithms 2 and 3: all dominated environments are removed from the label of each affected node. □

7.6 Experimental Results

We have developed a prototype implementation of our approach. Both the reasoner and the D-ATMS are implemented in Pop-11.[18] The tableaux reasoner is implemented as a set of six inference rules using Poprulebase, a Pop-11 rule interpreter.

To evaluate our approach, we performed experiments in which we compared the performance of our prototype system when providing all minimal explanations for inconsistencies in a variety of unfoldable and dictionary $\mathcal{ALC}$ TBoxes with that of MUPSter [82] and Pellet [86] (version 2.2.2). We chose to compare the D-ATMS with MUPSter and Pellet as they represent different approaches to finding all minimal explanations for an inconsistency. Both use a glass-box approach (extending the reasoner with dependency tracking), but MUPSter finds all minimal explanations, while Pellet finds a single minimal explanation, which is then combined with Reiter's Hitting Set algorithm [75] to find all other explanations [56, 86]. (In our experiments, we used Pellet's glass-box approach, as this typically requires less time to find an explanation [56].) The experiments were performed on a PC with dual quad-core 2.66GHz Intel Xeons and 32GB RAM PC running CentOS 5.5. All times are CPU times in ms and represent the average of 5 runs. Only the time actually used for generating explanations is given. We do not count the time AOD, MUPSter, and Pellet spend parsing and loading the ontologies, nor the time required for them to render the explanations.[19]

To test the correctness of our implementation, we compared the results for AOD with those of MUPSter on the set of 1,611 randomly generated unfoldable $\mathcal{ALC}$ TBoxes used by Schlobach to evaluate the performance of MUPSter [82].[20] For

[18]`http://www.cs.bham.ac.uk/research/projects/poplog/freepoplog.html`

[19]In addition we modified MUPSter so as not to require Racer, as the unsatisfiable concept is given as an input.

[20]The dataset is available at `http://www.few.vu.nl/~schlobac/software.html`.

each ontology, we obtained a list of unsatisfiable concept names from RacerPro before finding all minimal explanations for each unsatisfiable concept name.[21] The explanations generated by both systems were the same, apart from one case where MUPSter returned a non-minimal explanation.[22]

We also recorded the CPU time required for AOD, MUPSter and Pellet to generate explanations for each randomly generated ontology. In one case MUPSter did not produce an explanation within 5000 seconds and the run was aborted. We omitted this case and the case in which MUPSter returned a non-minimal explanation from our analysis, and in the following we consider only the remaining 1609 cases. Overall, AOD was noticeably faster than both MUPSter and Pellet, with an average execution time of 23 ms (median 6ms) compared to 671ms (median 36ms) for MUPSter and 192ms (median 162ms) for Pellet.

To evaluate the performance of AOD on more realistic examples, we used the Biochemistry-primitive ontology from the TONES repository,[23] a fragment of the Ordnance Survey BuildingsAndPlaces ontology,[24] and the Adult Mouse Brain Ontology from the NCBO BioPortal.[25] The Biochemistry-primitive, BuildingsAndPlaces, and Adult Mouse Brain ontologies were translated into $\mathcal{ALC}$ by removing axioms for inverse roles and role inclusions. To make these ontologies incoherent, we choose to systematically create unsatisfiable concepts from existing ontology entailments, allowing us to control the number of unsatisfiable concepts and the form of the resulting explanations. For each ontology, we randomly selected 10 pairs of concepts (A, B) where $A \sqsubseteq B$ is non-trivially entailed by the ontology, i.e., $A \sqsubseteq B \notin \mathcal{T}$. Then for each entailment, $A \sqsubseteq B$, we created

[21]`http://www.racer-systems.com/products/racerpro`

[22]For the TBox `tbox_50_6_1_1_3_5_v1` and unsatisfiable concept $A49$ MUPSter returns $\{A49, A37, A26, A34, A0\}$ as an explanation for the unsatisfiability of $A49$, while the D-ATMS returns $\{A49, A37, A34, A0\}$. As can easily be determined by hand (and confirmed by Pellet), the minimal set of axioms in addition to $A49(a)$ required to derive a contradiction in this case is indeed $\{A49, A37, A34, A0\}$.

[23]`http://owl.cs.manchester.ac.uk/repository`

[24]`http://www.ordnancesurvey.co.uk/oswebsite/ontology/BuildingsAndPlaces/v1.1/BuildingsAndPlaces.owl`

[25]`http://bioportal.bioontology.org/ontologies/1290`

a concept $EntailmentA\_B \sqsubseteq A \sqcap \neg B$. Finding all minimal explanations for the entailment $A \sqsubseteq B$ thus becomes equivalent to finding all minimal explanations for the unsatisfiability of $EntailmentA\_B$.

The explanations generated by all three systems were the same for all ontologies. The timing results are presented in Table 7.1, 7.2, and 7.3. As can be seen, AOD is 4.25 to 7.6 times faster than MUPSter and 14.5 to 23 times faster than Pellet on these ontologies (based on ratios of median times). Moreover, the maximum execution time of AOD for any concept is less than the minimum execution time of MUPSter and Pellet for any concept.

	AOD	MUPSter	Pellet
Average	8	52	144
Median	5	38	106
Min	5	36	101
Max	27	178	486

TABLE 7.1: Execution times (in ms) for the Biochemistry-primitive ontology (265 axioms and 10 unsatisifiable concepts).

	AOD	MUPSter	Pellet
Average	9	35	116
Median	8	34	116
Min	5	31	109
Max	15	40	125

TABLE 7.2: Execution times (in ms) for the BuildingsAndPlaces ontology (124 axioms and 10 unsatisifiable concepts).

	AOD	MUPSter	Pellet
Average	23	155	493
Median	21	161	492
Min	17	129	486
Max	27	178	503

TABLE 7.3: Execution times (in ms) for the Adult Mouse Brain ontology (3447 axioms and 10 unsatisifiable concepts).

We also evaluated the performance of AOD on four large-scale, cyclic ontologies: the Geo ontology [82], the DICE ontology[26], the MGED ontology[27], and a merge of two well-known upper ontologies, a mini-version of SUMO ontology[28] and the CYC ontology[29]. As in [82], the Geo, DICE, and MGED ontologies were made incoherent by adding disjointness axioms of the form $DJ(A_1, \dots, A_n)$ stating that the concepts $A_1, \dots, A_n$ are pairwise disjoint. For the merge of mini SUMO and CYC, the large number of concepts common to both ontologies results in many unsatisfiable concepts in the merged ontology.

The results are presented in Tables 7.4, 7.5, 7.6, and 7.7. For the merge of the mini SUMO and CYC ontologies in Table 7.7, we also recorded the number of concepts where MUPSter and Pellet did not return an explanation within 600 seconds. Where MUPSter and Pellet generated explanations, these were the same as those generated by AOD. As with the non-cyclic ontologies above, AOD is noticeably faster than both MUPSter and Pellet (5.4 to 54 times faster than MUPSter and 32 to approximately fifteen thousand times faster than Pellet). In addition, the maximum time required for AOD to return an explanation in the most complex ontology, the merge of mini SUMO and CYC,[30] is 18.2 seconds, whereas MUPSter fails to return explanations of three concepts within 600 seconds, and Pellet fails to return explanations for 255 concepts (27% of the total) within 600 seconds.

Overall, these results suggest AOD is noticeably faster than MUPSter and Pellet on both unfoldable and cyclic ontologies. For seven of the eight non-random ontologies, it is uniformly faster on all concepts, often by a significant margin. For large, complex ontologies, such as the merge of mini SUMO and CYC, the improvement in performance is most noticeable, suggesting that the D-ATMS

[26]http://www.mindswap.org/2005/debugging/ontologies/dice.owl
[27]http://www.mged.org
[28]http://www.ontologyportal.org
[29]http://www.opencyc.org
[30]While the DICE ontology contains a larger number of axioms, approximately 98% are disjointness axioms.

	AOD	MUPSter	Pellet
Average	7	52	289
Median	7	50	287
Min	5	48	239
Max	11	61	368

TABLE 7.4: Execution times (in ms) for the Geo ontology (500 axioms and 11 unsatisifiable concepts).

	AOD	MUPSter	Pellet
Average	10	524	75235
Median	9	489	75251
Min	5	359	74059
Max	19	649	76670

TABLE 7.5: Execution times (in ms) for the DICE ontology (27939 axioms and 76 unsatisifiable concepts).

	AOD	MUPSter	Pellet
Average	35	71	800
Median	15	68	485
Min	5	45	404
Max	217	146	8470

TABLE 7.6: Execution times (in ms) for the MGED ontology (406 axioms and 72 unsatisifiable concepts).

	AOD	MUPSter	Pellet
Average	132	3638	20658
Median	13	334	712
Min	5	257	592
Max	18167	494030	536422
Timeout after 600s	0	3	255

TABLE 7.7: Execution times (in ms) for the Mini Sumo & Cyc ontology (5725 axioms and 923 unsatisifiable concepts).

may be more scalable than MUPSter and Pellet. However, on the MGED ontology, although is AOD is about 5.4 times faster than MUPSter overall (based on the ratio of median times), it is slower than MUPSter on about 10% of concepts by a factor of up to 2. An analysis of profiling data for AOD suggests that for these concepts, execution time is dominated by the reasoner, with only a small amount of time spent in explanation generation. With a more sophisticated reasoner implemenation, we might expect AOD's execution time to reduce in such cases.

7.7 Displaying Explanations

As the D-ATMS maintains an explicit justification structure, it is straightforward to generate explanations of *how* a contradiction is derivable intended for human users. The D-ATMS keeps track of intermediate steps in a derivation as justifications in the justification graph and can render them in textual form for output to the user. For example, the explanations for the MadCow example in Section 7.4.3 (with minor editing to fit page margins) are:

$\perp$ **derived from:** $Animal(b)$, $\neg Animal(b)$

    $Animal(b)$ **derived from: axiom** $Sheep \sqsubseteq Animal$, $Sheep(b)$

        $Sheep(b)$ **derived from:** $(Sheep \sqcup Cow)(b)$

*

| $Animal(b)$ **derived from:** $(Animal \sqcap \forall\, eats.\neg Animal)(b)$

| $(Animal \sqcap \forall\, eats.\neg Animal)(b)$ **derived from:**

| **axiom** $Cow \sqsubseteq Animal \sqcap \forall\, eats.\neg Animal$, $Cow(b)$

| $Cow(b)$ **derived from:** $(Sheep \sqcup Cow)(b)$

|

* $(Sheep \sqcup Cow)(b)$ **derived from:** $\exists\, eats.(Sheep \sqcup Cow)(a))$

                $\exists\, eats.(Sheep \sqcup Cow)(a)$ **derived from:** $Cow(a) \sqcap \exists\, eats.(Sheep \sqcup Cow)(a)$

                    $Cow(a) \sqcap \exists\, eats.(Sheep \sqcup Cow)(a)$ **derived from:**

                    axiom $MadCow \sqsubseteq Cow \sqcap \exists\, eats.(Sheep \sqcup Cow)$, **axiom** $MadCow(a)$

    $\neg Animal(b)$ **derived from:** $\forall\, eats.\neg Animal(a)$, $eats(a, b)$

$\forall\, eats.\neg Animal(a)$ **derived from:** $Animal \sqcap \forall\, eats.\neg Animal(a)$

$Animal \sqcap \forall\, eats.\neg Animal(a)$ **derived from:**

axiom $Cow \sqsubseteq Animal \sqcap \forall\, eats.\neg Animal$, $Cow(a)$

$Cow(a)$ **derived from:** $Cow(a) \sqcap \exists\, eats.(Sheep \sqcup Cow)(a)$

$Cow(a) \sqcap \forall\, eats.(Sheep \sqcup Cow)(a)$ **derived from:**

axiom $MadCow \sqsubseteq Cow \sqcap \exists\, eats.(Sheep \sqcup Cow)$, **axiom** $MadCow(a)$

$eats(a, b)$ **derived from:** $\exists\, eats.(Sheep \sqcup Cow)(a)$

$\exists\, eats.(Sheep \sqcup Cow)(a)$ **derived from:** $Cow(a) \sqcap \exists\, eats.(Sheep \sqcup Cow)(a)$

$Cow(a) \sqcap \exists\, eats.(Sheep \sqcup Cow)(a)$ **derived from:**

axiom $MadCow \sqsubseteq Cow \sqcap \exists\, eats.(Sheep \sqcup Cow)$, **axiom** $MadCow(a)$

The explanation rendering of AOD is an initial prototype, and (as in the example above) the textual explanations produced are often rather verbose. There has been considerable work in the literature on the generating more 'human readable' explanations and other debugging aids (such as suggesting repairs to an ontology) [10] which we believe can be adapted in a straightforward way to the AOD justification structure.

7.8 Conclusion

We described AOD, a system for debugging dictionary $\mathcal{ALC}$ TBoxes based on an ATMS with disjunctions and loops. Our approach is correct and complete with respect to a reasoner for $\mathcal{ALC}$ with dictionary TBoxes. We also proposed a new blocking condition to ensure termination during the reasoning process. We presented experimental results which suggest that its performance compares favourably with that of MUPSter and Pellet. As the D-ATMS maintains an explicit justification structure, it is straightforward to generate explanations of *how* a contradiction is derivable intended for human users — the D-ATMS essentially keeps intermediate steps in a derivation and can produce them on request.

We believe the D-ATMS is a promising new approach to ontology debugging. Although our approach was developed for $\mathcal{ALC}$ with dictionary TBoxes, the reasoner and the reason maintenance component are only loosely coupled, and the D-ATMS can be adapted to work with other reasoners.

Chapter 8

Axiom Pinpointing for SUMO

8.1 Introduction

An upper ontology provides definitions of generic or abstract concepts that span a broad range of domain areas. Upper ontologies allow application developers to define new (domain specific) concepts in terms of a common ontology, and provide semantic interoperability by allowing applications to inter-operate through shared concepts. One such upper ontology is the Suggested Upper Merged Ontology (SUMO) [69]. SUMO contains about 1000 terms and 4000 definitional statements[1] expressed in a variant of first order logic with some higher-order extensions called the Standard Upper Ontology Knowledge Interchange format (SUO-KIF) [46].

In order for ontologies such as SUMO to be widely used, it is important to be able to guarantee that they are consistent and free of bugs. A 'bug' in the context of ontology development is the derivability of an undesirable consequence

[1]The term 'SUMO' is used to refer both to the upper ontology and to a collection of domain specific ontologies comprising about 20k terms based on the upper ontology. In what follows, we take SUMO to refer to the upper ontology only.

from the axioms of the ontology. *Explanation generation* is the process of providing human-understandable reasons for the derivability of some (typically undesirable) formula in an ontology. For logics which include classical propositional logic, it is sufficient to be able to provide an explanation for derivability of a contradiction. Namely, if we are interested in an explanation for the derivability of ϕ from Γ, we can reduce this problem to an explanation of derivability of a contradiction $\perp$ from $\Gamma \cup \{\neg\phi\}$. Explanations are key to debugging ontologies — when an undesired formula is derivable, it is important to know why it is derivable, so that the responsible axioms can be changed by the ontology developer. Several different styles of explanation generation have been proposed in the literature. One approach involves producing an (edited) proof trace, e.g., [36] (existing debugging tools for SUMO, e.g., [50, 70, 72] also fall into this category). Another approach is *axiom pinpointing*, i.e., the identification of the minimal set of ontology axioms from which a contradiction is derivable, e.g., [81, 86]. The reason for requiring a *minimal* set of axioms rather than the set of all axioms involved in a derivation, is because a derivation may contain redundant axioms, which makes it difficult to decide which axioms have to be removed or edited. In particular, proof traces may contain redundant steps and references to axioms which are not essentially used in the derivation.

This chapter presents an approach to axiom pinpointing for SUMO ontologies, SES (SUMO Explanation Service), that returns the set of minimal sets of ontology axioms from which a contradiction is derivable. SES consists of two parts: a second-order reasoner for SUO-KIF, and a truth maintenance system. The reasoner is sound and complete for a fragment of SUO-KIF in which all the SUMO Base ontology axioms can be expressed. The truth maintenance system computes explanations from the inferences made by the reasoner. The fragment of SUO-KIF understood by the reasoner is not decidable, and SES is therefore not guaranteed to produce all explanations for the derivability of $\perp$. However, when the reasoner does terminate 'naturally' (not due to a time-out) it is guaranteed to return all minimal explanations for $\bot$. Otherwise it returns all sets of axioms responsible for the derivations of $\bot$ it found before termination, minimised with respect to the inferences it found so far.

The research questions, objectives, and the contributions of the work presented in this chapter are as follows.

Research Questions How can the D-ATMS be extended to deal with logics where a decidable reasoning procedure does not exist? In case of non-termination, what should the D-ATMS-based explanation service do?

Research Objectives

1. To select a real-world use case where the ontology is reasonably large and the underlying logic is undecidable.
2. To implement a translating procedure from such logics to a logic which the D-ATMS can handle.
3. To implement an extension of the D-ATMS to provide an axiom pinpointing service for the ontology represented in the new logic as well as the special treatment in the case of non-termination.

Contributions The main contribution of this chapter is the SES, an approach to axiom pinpointing for SUMO ontologies, which returns the set of minimal sets of ontology axioms from which a contradiction is derivable. To the best of our knowledge, SES is the first system to provide axiom pinpointing-style explanations for SUMO ontologies. Another contribution is the FKIF logic, a fragment of second-order logic which allows efficient implementation of the reasoner while still is able to represent most axioms in the SUMO Base Ontology.

The remainder of this chapter is organized as follows. In sections 8.2 and 8.3 we briefly outline SUMO and SUO-KIF, and the fragment of second-order logic FKIF understood by the reasoner. In Section 8.4 we describe the two main components of our prototype axiom pinpointing system SES, and in Section 8.5 we give examples of two bugs in the SUMO Base ontology found by SES. In Section 8.6 we briefly describe related work, and conclude in Section 8.7.

8.2 SUMO & SUO-KIF

The Suggested Upper Merged Ontology (SUMO) [69] is a freely available, formal ontology of about 1000 terms and 4000 definitional statements. It consists of eleven sub-ontologies (Structural, Base, Set/Class Theory, Numeric, Temporal, Mereotopology, Graph, Measure, Processes, Objects and Qualities), of which the most important are Base and Structural (all of the other sub-ontologies include these two). SUMO has undergone more than ten years of development, and has been extended with a number of domain ontologies which together comprise some 20,000 terms and 80,000 axioms. It has been applied in a number of areas including Artificial Intelligence and linguistics.

SUMO has been extensively peer reviewed during development, and has been subjected to a certain degree of formal verification using automated theorem provers [50, 70, 72]. In particular, the Sigma environment for the development of SUMO ontologies [70] can be used with a number of different automatic theorem provers, including Vampire [78] and E [84] to check whether an ontology is consistent. This work identified a number of inconsistencies in SUMO which were rectified, and much of the recent work on SUMO and Sigma has focused on increasing coverage for specific applications, rather than investigating the properties of the core upper ontology. The SUMO web site[2] lists no known bugs in the upper ontology.

SUMO is described using a variant of first order logic with some higher order extensions, called the Standard Upper Ontology Knowledge Interchange format (SUO-KIF).[3]

$$
\begin{array}{ll}
sentence & ::= word \mid equation \mid relsent \mid \\
 & \quad logsent \mid quantsent \mid ?word \\
equation & ::= (=\ term\ term) \\
relsent & ::= (relword\ argument+) \\
logsent & ::= (not\ sentence) \mid \\
 & \quad (and\ sentence+) \mid (or\ sentence+) \mid \\
 & \quad (\Rightarrow\ sentence\ sentence) \mid \\
 & \quad (\Leftrightarrow\ sentence\ sentence) \\
quantsent & ::= (forall\ (variable+)\ sentence \mid \\
 & \quad (exists\ (variable+)\ sentence) \\
term & ::= variable \mid word \mid string \mid \\
 & \quad funterm \mid number \mid sentence \\
argument & ::= (sentence \mid term) \\
variable & ::= ?word \mid @word \\
string & ::= ``character*" \\
funterm & ::= (funword\ argument+) \\
relword & ::= word \mid variable \\
funword & ::= word \mid variable \\
number & ::= [-]\ digit+\ [.digit]\ [exponent] \\
exponent & ::= e\ [-]\ digit+
\end{array}
$$

FIGURE 8.1: BNF syntax for SUO-KIF

The BNF syntax for SUO-KIF is given in Figure 8.1 and is mostly self-explanatory. Restated in more conventional logical notation, it includes first- and second-predicates (for example, $\mathit{Divisible}$ as in $\mathit{Divisible}(0,0)$ is a first-order predicate and $\mathit{instance}$ as in $\mathit{instance}(\mathit{Divisible}, \mathit{ReflexiveRelation})$ is a second-order predicate), functional symbols, equality $=$, boolean connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), quantifiers $\forall$ (for all) and $\exists$ (exists). It has first-order and second-order variables and allows quantification over relational variables, for

[2]`www.ontologyportal.org`

[3]http://suo.ieee.org/SUO/KIF/suo-kif.html

example it is possible to say

$$\forall x \forall R(\mathit{instance}(R, \mathit{ReflexiveRelation}) \Rightarrow R(x, x))$$

The most unusual feature of SUO-KIF are row variables, $@ROW$, which range over finite sequences of arguments of arbitrary finite length; we will write it in the logical notation as $\bar{x}$: a list of variables x_i of arbitrary finite length. For example, it is possible to say

$$\exists R\, \exists \bar{x}\, R(\bar{x})$$

(there exists some relation and some sequence of elements such that R holds for this sequence). A detailed description of SUO-KIF can be found in [71].

Clearly, since SUO-KIF includes full first order logic with functional symbols and equality, reasoning in it is undecidable. Examining the upper ontology, it is difficult to identify some decidable fragment of first or second order logic into which it would fit: the quantifier prefixes of axioms are often of the form $\forall\,\exists$ which is undecidable with full first-order logic and at least one binary predicate [15], they do not conform to the definition of guarded [4] or packed [63] fragment of first-order or monadic second-order logic, etc.

8.3 FKIF

Since we were unable to find any decidable fragment of first- or second-order logic expressive enough to formalise SUMO, we chose to work in a fragment of second-order logic which we call FKIF. Although FKIF is not decidable, it allows for a reasonably efficient reasoner implementation (unlike the full SUO-KIF, which has been shown by Horrocks and Voronkov [50] to have a non recursively enumerable set of validities). The definition of FKIF is given below. Note that FKIF is different from the schema form proposed by Hayes and Menzel [46] to reduce complexity of SUO-KIF.

Recall that a formula is in negation normal form if all negations occur only in front of atomic formulas.

Definition 8.1. FKIF

FKIF is the fragment of SUO-KIF which contains formulas of the following form:

- ground formulas in negation normal form which do not contain quantifiers
- implications of the form

  $$\forall X_1 \ldots \forall X_n (\phi \Rightarrow \psi)$$

  where

  - $X_1, \ldots, X_n$ are all the free variables in $\phi \Rightarrow \psi$
  - all of $X_1, \ldots, X_n$ occur in ϕ
  - ϕ and ψ are in negation normal form
  - ϕ is built using only negations and conjunctions, and
  - ψ contains only negations, conjunctions, disjunctions and $\exists x$ where x is a first order variable (x is not $@ROW$ or $?REL$)

We omit universal quantifiers in FKIF formulas (free variables are assumed to be universally quantified). The main syntactic restriction of the FKIF fragment compared to full SUO-KIF, is that relational variables and $@ROW$ variables are only allowed to occur universally bound.

Not all well-formed SUO-KIF sentences can be equivalently translated into FKIF. For example

$$(exists\ (@ROW\ ?REL)\ (?REL\ @ROW))$$

(in logical notation, $\exists R\, \exists \bar{x}\, R(\bar{x})$) cannot be translated into FKIF. However there were no examples of this form in any SUMO ontology we examined.

The Base ontology was translated into FKIF using a translation procedure δ, which is applied recursively to a set of SUO-KIF sentences. The patterns matched by δ are given below, and are attempted in the order listed. We assume that each quantifier has its own distinct variable.

1. $\delta(\phi) = \phi$ if $\phi \in FKIF$
2. $\delta(\neg\exists\bar{X}\phi) = \forall\bar{X}\neg\phi$
3. $\delta(\forall\bar{X}\phi) = \exists\bar{X}\neg\phi \Rightarrow\perp$
4. $\delta(\exists\bar{X}\phi) = \top \Rightarrow \exists\bar{X}\phi$
5. $\delta(\phi_1 \Leftrightarrow \phi_2) = \{(\phi_1 \Rightarrow \phi_2), (\phi_2 \Rightarrow \phi_1)\}$
6. $\delta(\phi_1 \Rightarrow (\phi_2 \Rightarrow \phi_3)) = (\phi_1 \wedge \phi_2) \Rightarrow \phi_3$
   $\delta((\phi_1 \Rightarrow \phi_2) \Rightarrow \phi_3) = (\neg\phi_1 \vee \phi_2) \Rightarrow \phi_3$
7. $\delta(\phi_1 \Rightarrow \phi_2) = \mathit{dnf}(\phi_1) \Rightarrow \mathit{nnf}(\phi_2)$
8. $\delta(\phi_1 \Rightarrow (\forall\bar{X}\phi_2)) = (\exists\bar{X}(\phi_1 \wedge \neg\phi_2) \Rightarrow\perp$
9. $\delta((\exists\bar{X}\phi_1) \Rightarrow \phi_2) = \phi_1 \Rightarrow \phi_2$
10. $\delta((\forall\bar{X}\phi_1) \Rightarrow \phi_2) = \neg\phi_2 \Rightarrow \exists\bar{X}\neg\phi_1$
11. $\delta(\phi_1 \Rightarrow \phi_2) = \mathit{dnf}(\phi_1) \Rightarrow \mathit{nnf}(\phi_2)$

12. $\delta((\phi_1 \vee \phi_2) \Rightarrow \phi_3) = \{\phi_1 \Rightarrow \phi_3, \phi_2 \Rightarrow \phi_3\}$.

where *dnf* reduces a formula to disjunction normal form, and *nnf* reduces a formula to negation normal form.

Theorem 8.2. *If the translation procedure δ terminates, it is correct, i.e., the translated formula is in FKIF.*

Proof. Firstly, Steps 2, 3, 4, and 5 translate an existential or universal quantified statement into an implication. After that, Step 6 removes all nested implications. Next, Step 7 pushes all quantifiers to the front of the antecedent and the consequence of the implication. Steps 8, 9, 10 remove all quantifiers from the antecedent and the consequence of the implication where possible. However, Step 10 will not terminate if the input implication is of the form $\forall \bar{X} \phi_1 \Rightarrow \exists \bar{Y} \phi_2$. With such an input, the translated result is $\forall \bar{Y} \neg \phi_2 \Rightarrow \exists \neg \bar{X} \phi_1$, and hence Step 10 is repeated with the translated result.

However, if Step 10 terminates, Step 11 will translate the antecedent into a list of disjunctions. Finally, Step 12 will translate that implication into separate implications. □

One sentence in the Base ontology could not be translated by δ and required special treatment. An axiom defining the concept *TotalValuedRelation* contained a $@ROW$ variable which was 'underconstrained'. The axiom is given in its entirety in Section 8.5; here we give it in simplified form:

$$instance(R, TotalValuedRelation) \wedge Valence(R, n) \wedge$$
$$RightType(\bar{x}) \Rightarrow \exists y R(\bar{x}, y)$$

where $RightType(\bar{x})$ checks that $\bar{x}$ is of length $n-1$ and each element of it is in the appropriate domain for R. For example, a particular instance of this implication for $n = 3$ would be:

$$instance(Sum, TotalValuedRelation) \land Valence(Sum, 3) \land$$
$$Number(x_1) \land Number(x_2) \Rightarrow \exists y Sum(x_1, x_2, y)$$

One way of translating such axioms would be to introduce an instance of the axiom for each possible arity of the relation R, up to a specified bound determined by the current ontology.[4] SES takes a different approach, which involves a single rule that dynamically matches a set of instances for $\bar{x}$ once the relation's valence and type of elements are known. The final pattern for $@ROW$ in the reasoner rule that implements the axiom is a 'dynamic pattern' (effectively a procedure) that takes the valence of the relation and the element type, and returns $n-1$ previously asserted instances of the correct type.

8.4 Axiom Pinpointing for SUMO

Our implementation of axiom pinpointing for SUMO (SES) consists of two parts: a second-order reasoner for FKIF which derives consequences by applying inference rules to previously inferred sentences, and a truth maintenance system which maintains dependencies between newly inferred consequences and their antecedents and which computes explanations when a new derivation is found for $\perp$.

[4]This is the approach taken by Sigma, which encodes the above axiom as six instances to handle cases up to arity 6.

The system works in cycles. At each cycle, the reasoner checks whether any of its inference rules (described below) are applicable to the axioms and/or any previously inferred sentences, and if so, it sends the consequence of applying the rule together with a justification consisting of the name of the rule and the antecedents of the rule used to infer the consequence to the truth maintenance system. The truth maintenance system updates its dependency structure to incorporate the new formula and its justification. If a new closed derivation of $\perp$ has been found, the truth maintenance system also updates the set of explanations to record the minimal sets of ontology axioms required for all derivations of $\perp$ produced by the reasoner up to this point.

The system can be configured to pause after each new explanation is found, allowing the user to decide whether to continue or to terminate the search for explanations, or to run for a fixed number of reasoner cycles and return all explanations found within the depth bound.

Both the reasoner and the truth maintenance system are implemented in Pop-11.[5] The reasoner is implemented using Poprulebase, a Pop-11 rule interpreter which supports dynamic generation of rule patterns (used to implement rule conditions involving $@ROW$ variables) .

8.4.1 Reasoner

The reasoner uses the following tableau decomposition rules:

$\Rightarrow$**-rule** from $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi$ and $\phi_1[\bar{X}/\bar{t}], \ldots, \phi_n[\bar{X}/\bar{t}]$ derive $\psi[\bar{X}/\bar{t}]$, where $\phi_i[\bar{X}/\bar{t}]$ are ground instances of ϕ_i;

$\wedge$**-rule** from $\phi_1 \wedge \ldots \wedge \phi_n$ derive $\phi_1, \ldots, \phi_n$;

$\vee$**-rule** from $\phi_1 \vee \ldots \vee \phi_n$, derive cases ϕ_1, ..., ϕ_n;

[5] `http://www.cs.bham.ac.uk/research/projects/poplog/freepoplog.html`

$\exists$-rule from $\exists x \phi(x)$ derive $\phi(c)$ where c is a new individual and $\exists x \phi(x)$ has not been used before to generate another new individual;

$\perp$-rule from ϕ and $\neg\phi$ derive $\perp$, where ϕ is a ground atomic formula.

We assume semantics for SUO-KIF described in [46].

Theorem 8.3. *Each deterministic rule preserves satisfiability. If the premise of the non-deterministic rule $\vee$-rule is satisfiable, then one of the conclusions is.*

Proof. The rules for $\wedge, \vee, \exists, \perp$ (note that $\exists$ only applies to first order variables), are standard tableau rules. The only unusual rule is $\Rightarrow$. It treats implication not as standard classical tableaux do (splitting into cases) but as for example subsumption axioms are treated in description logic. The universally quantified variables which can be both first- and second-order, are instantiated against the domain as it is defined in [46]. Clearly, if the implication is valid and the left-hand side of it is satisfiable for some substitution of ground terms, then the right-hand side should be satisfiable as well. $\square$

Theorem 8.4. *Consider a tableau for a set of formulas Γ. If it has a branch where a contradiction is not derivable, then Γ is satisfiable.*

Proof. The proof is by constructing a satisfying model for Γ given an open branch of a standard tableau (a branch not containing a contradiction). A set of formulas Σ is a Hintikka set if:

1. for no formula ϕ both ϕ and $\neg\phi$ are in Σ
2. if $\phi \wedge \psi \in \Sigma$ then $\phi \in \Sigma$ and $\psi \in \Sigma$
3. if $\phi \vee \psi \in \Sigma$ then $\phi \in \Sigma$ or $\psi \in \Sigma$
4. if $\exists x \phi \in \Sigma$, then $\phi(c) \in \Sigma$ for some c

5. if $\forall \bar{X}(\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi) \in \Sigma$, and some ground instances of $\phi_1, \ldots, \phi_n$, $\phi_1[\bar{X}/\bar{t}], \ldots, \phi_n[\bar{X}/\bar{t}] \in \Sigma$, then $\psi[\bar{X}/\bar{t}] \in \Sigma$.

Clearly an open branch is a Hintikka set, and a Hintikka set is satisfiable. Note that when constructing a model, we can replace implications containing universally quantified variables with the set of all their ground instances. $\square$

Like any other first-order theorem prover, the reasoner is not guaranteed to terminate. We will say that the reasoner 'terminates normally' when it terminates because no new inferences can be found. To refer to both the cases when the reasoner terminates normally and when it times out, we will use the term 'halted'.

8.4.2 Truth Maintenance System

The D-ATMS truth maintenance system described in the previous chapter used by SES is essentially an Assumption Based Truth Maintenance System (ATMS) [23] extended to handle disjunctions.

The D-ATMS maintains a graph data structure which records all inference rule applications. Each derived formula ϕ is represented by a node n_ϕ. Axioms are represented by axiom nodes, and inconsistency is represented by a distinguished false node, $n_\perp$. Justifications form the edges of the graph and record the fact that a node (the consequent) can be derived from a set of other nodes (the antecedents). A node may be the consequent of more than one justification (recording the different ways in which it can be derived), and be an antecedent in other justifications (recording the inferences that can be made using it). When reasoning begins, the D-ATMS contains only axiom nodes. As the reasoner derives consequences, it sends the inferences to the D-ATMS. A justification is added linking the nodes representing the antecedents of the inference to the node representing the consequent (if no node exists for the derived formula, one is created). The reasoner may designate certain sets of formulas as inconsistent by providing a justification for $n_\perp$.

The derivability of a formula from a set of axioms is represented by an environment. Each node in the justification graph has a label containing the set of environments from which the formula corresponding to the node can be derived. The label of $n_\perp$ consists of a set of inconsistent environments or nogoods. The D-ATMS ensures that the environments in the label of each node are sound, complete and minimal with respect to the set of inferences passed by the reasoner to the D-ATMS so far.

The notion of *relative* correctness and minimality (with respect to the reasoner) is essential. The D-ATMS itself does not produce derivations; it only manipulates the inferences provided by the reasoner. If the reasoner's inference rules are unsound, then some environment may contain a set of axioms that do not logically entail the formula. If the set of inference rules used by the reasoner is incomplete for a given logic, then the environments in the label of a formula node may not contain all the sets of axioms that logically entail it. The environments generated by the D-ATMS are guaranteed to be minimal, but minimality is also relative to the set of rules used by the reasoner, and to the set of derivations discovered by the reasoner when the environment is (re)computed.

For example, suppose that the reasoner sends an inference to the D-ATMS recording that ϕ can be derived from ψ_1 and ψ_2. Suppose further that this is the first derivation of ϕ which has been found and that the labels of ψ_1 and ψ_2 have one environment each: ψ_1 is ultimately derivable from the axioms $\{ax_1, ax_2\}$ and ψ_2 is derivable from $\{ax_2, ax_3\}$. Then the single environment in the label of n_ϕ will be $\{ax_1, ax_2, ax_3\}$. Given the set of inferences made by the reasoner, this is a minimal environment. However, suppose that at the next step the reasoner discovers another derivation of ϕ, this time from ψ_2 and ψ_3 which has a single environment $\{ax_3\}$. When this inference is passed to the D-ATMS, the label of n_ϕ is updated to include the environment $\{ax_2, ax_3\}$; since the new environment is a subset of the old environment $\{ax_1, ax_2, ax_3\}$, $\{ax_1, ax_2, ax_3\}$ is discarded to maintain mimimality of the label of n_ϕ. The label of n_ϕ would be updated in the same way if the reasoner discovers that ψ_1 is derivable just from only $\{ax_2\}$, as changes in environments are propagated to the 'descendants' of the formula: n_ϕ's environment will become $\{ax_2, ax_3\}$.

The D-ATMS label computation algorithms are correct in the following sense:

D-ATMS Correctness. When the reasoner is halted, the D-ATMS returns as the label of $n_\perp$, a set of sets of axioms $\{e_1, \ldots, e_n\}$ such that each set e_i is a set of axioms for which the reasoner found a derivation of $\perp$, and this set is minimal with respect to all inferences found before the reasoner halted.

Our argument in the previous chapter (e.g., Section 7.5) builds on standard ATMS results and uses the fact that the D-ATMS algorithms ensure that each environment in a label of a formula involved in a derivation of $\perp$ contains a minimal set of axioms required to derive the formula (relative to the existing justification graph). Note that if the reasoner terminates normally (when no more inference rules are applicable), SES will return all possible explanations (minimal sets of axioms responsible for inconsistency). If the reasoner terminates due to e.g. time out, the explanations it returns are still guaranteed to be correct (the axioms in each explanation do entail false) but they not guaranteed to be minimal, nor is SES guaranteed to return all possible explanations in this case.

8.5 Examples

To illustrate the utility of our approach, we give two examples of bugs in the Base ontology of SUMO discovered by SES. We chose the Base ontology as it is reasonably large (consisting of 1058 sentences), complex, and is used by many other ontologies. We found two different derivations of $\perp$ using two axioms which have the same bug (a missing universal quantifier).

The simpler of the two bugs is in the definition of a reflexive relation:

```
(<=>
 (instance ?REL ReflexiveRelation)
 (?REL ?INST ?INST))
```

which is missing a (forall ?INST) quantifier on the right. It is translated by δ into two FKIF axioms

$$\forall r \forall i_1 \forall i_2 (\mathit{instance}(r, \mathit{ReflexiveRelation}) \Rightarrow r(i_1, i_2))$$

and

$$\forall r \forall i_1 \forall i_2 (r(i_1, i_2) \Rightarrow \mathit{instance}(r, \mathit{ReflexiveRelation}))$$

From the Base ontology and two additional facts, $\mathit{Divisible}(1,1)$ and $\neg \mathit{Divisible}(0,0)$, SES derives a contradiction and gives as an explanation the two facts and the definition of the reflexive relation.

The second example of an inconsistency involves the definition of a total valued relation

```
(<=>
 (instance ?REL TotalValuedRelation)
 (exists (?VALENCE)
  (and
   (instance ?REL Relation)
   (valence ?REL ?VALENCE)
   (=>
    (forall (?NUMBER ?ELEMENT ?CLASS)
     (=>
      (and
       (lessThan ?NUMBER ?VALENCE)
       (domain ?REL ?NUMBER ?CLASS)
       (equal
        ?ELEMENT
        (ListOrderFn (ListFn @ROW) ?NUMBER)))
      (instance ?ELEMENT ?CLASS)))
    (exists (?ITEM)
     (?REL @ROW ?ITEM)))))
```

The definition of $\mathit{TotalValuedRelation}$ also has a missing universal quantifier: in this case over the $@ROW$ variable on the right hand side. The problematic direction is $\Leftarrow$, which essentially says that if there exists a single tuple of the correct type satisfying $?REL$, then $?REL$ is a total valued relation:

$$R(\bar{x}, y) \Rightarrow \mathit{instance}(R, \mathit{TotalValuedRelation}) \wedge$$
$$\mathit{Valence}(R, n) \wedge \mathit{RightType}(\bar{x})$$

Both derivations of $\perp$ were found within a few seconds. However, the reasoner does not terminate naturally on the base ABox, hence we have no guarantee that no other derivations of $\perp$ exist.[6]

[6]An analysis of the performance of the reasoner reveals that the reason for non-termination on the base ABox is generation of new terms by function application. While it is possible a add a blocking condition which blocks further application of reasoner rules when some limit on the nesting of functions is exceeded, this would result in loss of completeness.

We checked the derivation of inconsistency for the $\mathit{ReflexiveRelation}$ using Sigma [70]. Sigma can derive $\mathit{Divisible}(0, 0)$ as an answer to a query (a request to prove $\mathit{Divisible}(?X?X)$), and it correctly states that the derivation of $\mathit{Divisible}(0, 0)$ indicates an inconsistency. However, when we used Sigma to check consistency of the Base ontology (as opposed to answering a query with a concrete predicate), it ran out of memory even with a heap size of 10GB.

8.6 Related work

Horrocks and Voronkov [50] used the first-order theorem prover Vampire [78] for query answering and consistency checking in SUMO. They discovered a number of non-trivial inconsistencies. As an explanation of an inconsistency, they give an (edited) proof listing. They comment on the problem of making proofs human-readable and understandable, and concede that current proof format of Vampire is far from perfect. The Sigma ontology development environment for SUMO [70, 72] can be used for query answering and inconsistency checking, however in our experience the proof listings are somewhat difficult to understand.

8.7 Conclusion

We described SES, an approach to axiom pinpointing for SUMO ontologies, which returns the set of minimal sets of ontology axioms from which a contradiction is derivable. SES consists of two main components: a second-order tableaux reasoner, and a truth maintenance system. The reasoner is sound and complete for a fragment of SUO-KIF in which all the SUMO Base ontology axioms can be expressed. The truth maintenance system computes explanations from the inferences made by the reasoner. The combined returns all sets of axioms responsible for the derivations of $\perp$ found before termination, minimised with respect to the inferences it found so far. In cases where SES does not time out, it is guaranteed to return all minimal explanations for $\perp$.

To the best of our knowledge, SES is the first system to provide axiom pinpointing-style explanations for SUMO ontologies. Although the reasoner can be further optimised, our prototype SES implementation is able to debug the SUMO Base ontology, finding two previously unreported derivations of $\perp$ in a few seconds.

Chapter 9

Conclusion and Future Work

9.1 Summary of Contributions

In this thesis, we have shown that TMS can be used in modern Knowledge-Based Systems such as intelligent agents and ontologies.

Firstly, we showed that the dependency tracking mechanism in TMS can be used in agent programming platforms, and not only for belief revision as in the literature [3, 52, 61], but also for improving performance of agent programs. In Chapter 5, we applied a light-weight version of a TMS to keep track of the dependency between facts and queries in the agent databases so that if there is an update in the agent databases, it is possible to find the affected queries. Using this system, we were able to perform query caching in the GOAL agent programming language, following the observations given in [2]. The caching mode can be either single cycle, i.e., the cache is cleared after a query-update cycle, or multi-cycle, i.e., the cache is maintained over multiple query-update cycles. Our approach supports the multi-cycle caching mode and only removes from the cache queries' results invalidated after an update. The experiments in different caching modes showed that query caching improves the performance of agent programs and multi-cycle caching performs better than single-cycle caching in all test-cases.

Secondly, we argued that it is possible to apply the ATMS to the debugging/axiom pinpointing problems in ontologies with different levels of expressiveness. In fact, previous glass-box approaches to the problem of ontology debugging/axiom pinpointing use a tracing facility embedded inside the reasoner to record the dependencies between the derived data (assertion) and the original assumptions (axioms) used to derive it [57, 59, 64, 81]. Therefore, the dependency tracking facility and the reasoner are tightly-coupled, and hence for each implementation of a reasoner, the facility for axiom pinpointing is built from scratch. On the other hand, the ATMS and the reasoner is loosely-coupled. For example, in Chapter 6, we have showed that the "classic" ATMS can directly deal with the ontology debugging/axiom pinpointing problem when the reasoner is a basic forward chaining inference engine with only Horn-like rules. As long as the reasoner has only Horn-like rules and there is no cycle obtained by generating new constants, the ATMS-based approach presented in Chapter 6 will work for the ontology debugging/axiom pinpointing problem.

For logics which have disjunctions such as the description logic $\mathcal{ALC}$, the ATMS needs to be extended to deal with disjunctions, as in Chapter 7. Disjunctions in the ATMS are solved by recording the sequence of choices in each environment so that the extended environment has not only the set of assumptions where a datum holds but also the sequence of choices which have been made to derive the datum. For ontologies which have cyclic inclusions, a blocking condition is necessary to guarantee termination. However, to allow node labels in the ATMS to be complete, the blocking condition needs to take into account not only the node's datum, i.e., the assertion, but also its label. In Chapter 7, we have presented a blocking condition for a Dictionary $\mathcal{ALC}$ reasoner which can guarantee completeness for the ATMS node labels relative to this reasoner. In this chapter, we also showed how the label update propagation can be optimised by focusing on relevant justifications, i.e., justifications involved in the derivation of a target node. The results of the experiments comparing our approach and two Description Logics reasoners, Pellet and MUPSter, suggest that the ATMS-based approach outperforms two other systems in a wide range of ontologies.

For more expressive logics, in Chapter 8 we investigated whether the extended ATMS can find explanations for a contradiction in an upper ontology, SUMO, whose underlying logic, SUO-KIF, includes full first order logic and some higher-order features. To be able to use the ATMS for debugging SUMO, we defined a second-order fragment of SUO-KIF, namely FKIF, and showed how to transform a formula from SUO-KIF to FKIF using a translation procedure. We were able to translate most statements in the SUMO's Base ontology to FKIF, apart from one statement which needs special treatment. The reasoner for FKIF is implemented and, combined with the extended ATMS, form an explanation service for SUMO, which we call SES. As the reasoner is not guaranteed to terminate, we configured the ATMS so that it can either compute all minimal explanations for inconsistency if the reasoner terminates, or return possibly incomplete and non-minimal explanations if the reasoner halts due to time out or reaching the bound depth. To the best of our knowledge, SES is the first system to provide axiom pinpointing-style of explanations for SUMO.

9.2 Future Work

This thesis has showed that it is possible to use the ATMS to find explanations of a derivation or an inconsistency in ontology-based systems. In future, our ATMS-based explanation framework can be extended in two aspects, generalisation and efficiency.

Regarding generalisation, a future plan is to characterise the types of reasoning procedures which the ATMS can be used to to provide an explanation service. For example, as showed in [9], a terminating tableaux-based reasoner does not always have a terminating axiom pinpointing extension. Therefore, important topics for future work is to characterise such reasoners and to use the ATMS to provide possibly incomplete but terminating axiom pinpointing service. This is similar to how we treated the SUMO upper ontology in Chapter 8.

To improve efficiency of our current framework, there are two potential directions. The first is to improve the performance of the reasoner implementation. The Poprulebase reasoners in our framework use a simple pattern-directed rule matching, with backtracking to find consistent variable bindings. While the underlying Poprulebase implementation does incorporate hashing of axioms and derived formulas, it does not make use of more sophisticated caching strategies such as, e.g., RETE [39]. As the reasoner must run until no rule is applicable to ensure all minimal explanations are found, in cases where very large numbers of inferences are possible, the lack of more sophisticated indexing/caching may have a significant impact. Secondly, the ATMS implementation can be optimised further by having a more efficient subsumption testing using extra data structures such as, e.g., tries as suggested in [38]. However, more work needs to be done for using tries for the extended ATMS, where disjunctions are allowed. In addition, for an acyclic dependency graph, a MapReduce technique as presented in [93] can also be used to improve the performance of label update propagation in the ATMS.

Although query caching in agent programs, especially multi-cycle query caching, has been showed to be able to improve the performance of agent programs in Chapter 5, we are still aware of several possible limitations of the current implementation which can be improved in future work. The most important topic of future work is to generalise the current implementation of query caching.

The current prototype of query caching is specific to the GOAL agent programming language and is also tied to a specific KRT, SWI-Prolog. This implementation takes advantage of the meta-programming feature in Prolog-based KRTs to record dependencies between queries and facts. Therefore, a promising research direction is to develop the general interface between the query caching component and the agent program as well as the KRT so that different combinations of agent programming platforms and KRTs can use query caching. With such an interface, the implementation of query caching component can be loosely-coupled to the KRT and to the implementation of the agent programming platforms.

Bibliography

[1] N. Alechina, T. Behrens, M. Dastani, K. Hindriks, J. Hubner, B. Logan, H. Nguyen, and M. van Zee. Multi-cycle query caching in agent programming. In *Proceedings of the Twenty-Seventh AAAI Conferernce on Artificial Intelligence (AAAI 2013)*, Bellevue, Washington, July 2013. AAAI, AAAI Press.

[2] N. Alechina, T. Behrens, K. V. Hindriks, and B. Logan. Query caching in agent programming languages. In M. Dastani, B. Logan, and J. F. Hubner, editors, *Proceedings of the Tenth International Workshop on Programming Multi-Agent Systems (ProMAS 2012)*, pages 117–131, Valencia, Spain, 06/2012 2012.

[3] N. Alechina, M. Jago, and B. Logan. Resource-bounded belief revision and contraction. In M. Baldoni, U. Endriss, A. Omicini, and P. Torroni, editors, *Proceedings of the Third International Workshop on Declarative Agent Languages and Technologies (DALT 2005)*, pages 118–131, Utrecht, July 2005.

[4] H. Andréka, J. van Benthem, and I. Németi. Modal logics and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[6] F. Baader, D. Calvanese, D. L. Mcguinness, D. Nardi, and P. F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2007.

[7] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In H. Boley and M. Richter, editors, *Processing Declarative Knowledge*, volume 567 of *Lecture Notes in Computer Science*, pages 67–86. Springer Berlin / Heidelberg, 1991.

[8] F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *Description Logic Handbook*, pages 47–100. Cambridge University Press, 2002.

[9] F. Baader and R. Peñaloza. Axiom pinpointing in general tableaux. *Journal of Logic and Computation*, 20(1):5–34, 2010. Special Issue: Tableaux and Analytic Proof Methods.

[10] S. Bail, M. Horridge, B. Parsia, and U. Sattler. The justificatory structure of the ncbo bioportal ontologies. In *ISWC 2011 - Proceedings of the 10th International Semantic Web Conference (ISWC 2011)*, volume 7031 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2011.

[11] T. M. Behrens, J. Dix, J. Hübner, and M. Köster. Special issue: The multi-agent programming contest: Environment interface and contestants in 2010, editorial. *Annals of Mathematics and Artificial Intelligence*, 61(4):257–260, 2011.

[12] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

[13] R. H. Bordini, L. Braubach, M. Dastani, A. E. Fallah-Seghrouchni, J. J. GÃşmez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica (Slovenia)*, 30(1):33–44, 2006.

[14] R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007.

[15] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[16] R. Brachman and H. Levesque. *Knowledge Representation and Reasoning*. Morgan Kaufmann, May 2004.

[17] J. Broekstra and A. Kampman. Inferencing and truth maintenance in RDF schema. In R. Volz, S. Decker, and I. F. Cruz, editors, *PSSS1 - Practical and Scalable Semantic Systems, Proceedings of the First International Workshop on Practical and Scalable Semantic Systems, Sanibel Island, Florida, USA, October 20, 2003*, volume 89 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[18] M. Buchheit, F. M. Donini, and A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *J. Artif. Intell. Res. (JAIR)*, 1:109–138, 1993.

[19] P. Dague. Model-based diagnosis of analog electronic circuits. *Annals of Mathematics and Artificial Intelligence*, 11(1-4):439–492, 1994.

[20] M. Dastani. 2APL: a practical agent programming language. *Journal of Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[21] M. Dastani, F. de Boer, F. Dignum, and J.-J. Meyer. Programming agent deliberation: an approach illustrated using the 3APL language. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*, pages 97–104, New York, NY, USA, 2003. ACM Press.

[22] M. Dastani, J. Dix, and P. Novak. The first contest on multi-agent systems based on computational logic. In F. Toni and P. Torroni, editors, *Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI,*

London, UK, June 27-29, 2005, Revised Selected and Invited Papers, pages 373–384. Springer, 2006.

[23] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[24] J. de Kleer. Extending the ATMS. *Artificial Intelligence*, 28(2):163–196, 1986.

[25] J. de Kleer. A General Labeling Algorithm for Assumption-Based Truth Maintenance. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI'88)*, pages 188–192. AAAI Press / The MIT Press, 1988.

[26] J. de Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, pages 780–785. The AAAI Press / The MIT Press, 1992.

[27] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97 – 130, 1987.

[28] J. de Kleer and B. C. Williams. Diagnosis with behavioral modes. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 1324–1330. Morgan Kaufmann, 1989.

[29] S. Dixon and N. Foo. Connections between the ATMS and AGM belief revision. In *Proceedings of the 13th international joint conference on Artifical intelligence - Volume 1*, IJCAI'93, pages 534–539, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[30] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[31] O. Dressler and A. Farquhar. Putting the problem solver back in the driver's seat: Contextual control of the ATMS. In J. P. Martins and M. Reinfrank, editors, *Proceedings of the ECAI'90 Workshop on Truth Maintenance Systems*, 1990.

[32] O. Dressler and P. Struss. Model-based diagnosis with the default-based diagnosis engine: Effective control strategies that work in practice. In *ECAI'94*, pages 677–681, 1994.

[33] H. Du, N. Alechina, K. Stock, and M. Jackson. The logic of NEAR and FAR. In *Conference On Spatial Information Theory*, Lecture Notes in Computer Science. Springer, 2013.

[34] H. Du, S. Anand, N. Alechina, J. G. Morley, G. Hart, D. G. Leibovici, M. Jackson, and J. M. Ware. Geospatial information integration for authoritative and crowd sourced road vector data. *T. GIS*, 16(4):455–476, 2012.

[35] Elevator Simulator. `http://sourceforge.net/projects/elevatorsim/`, 2011.

[36] C. G. Fernandes, V. Furtado, A. Glass, and D. L. McGuinness. Towards the generation of explanations for semantic web services in OWL-S. In R. L. Wainwright and H. Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 2350–2351. ACM, 2008.

[37] K. D. Forbus and J. de Kleer. Focusing the ATMS. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 193–198. American Association for Artificial Intelligence, 1988.

[38] K. D. Forbus and J. de Kleer. *Building Problem Solvers*. MIT Press, Cambridge, MA, 1993.

[39] C. Forgy. RETE: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[40] M. P. Georgeff and A. L. Lansky. Reactive reasoning and planning. In *Proceedings of the sixth National conference on Artificial Intelligence (AAAI'87)*, AAAI'87, pages 677–682. AAAI Press, 1987.

[41] B. C. Grau, B. Parsia, E. Sirin, and A. Kalyanpur. Modularity and web ontologies. In P. Doherty, J. Mylopoulos, and C. A. Welty, editors, *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006*, pages 198–209. AAAI Press, 2006.

[42] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 9(3):508–511, 1997.

[43] Y. Guo and J. Heflin. An initial investigation into querying an untrustworthy and inconsistent web. In J. Golbeck, P. A. Bonatti, W. Nejdl, D. Olmedilla, and M. Winslett, editors, *Proceedings of the ISWC*04 Workshop on Trust, Security, and Reputation on the Semantic Web, Hiroshima, Japan, November 7, 2004*, volume 127 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.

[44] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, 1995.

[45] V. Haarslev and R. Möller. Racer: An OWL reasoning agent for the semantic web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence*, pages 91–95, 2003.

[46] P. Hayes and C. Menzel. A semantics for the knowledge interchange format. In *Proceedings of the IJCAI 2001 Workshop on the IEEE Standard Upper Ontology*, 2001.

[47] K. Hindriks, F. Boer, W. Hoek, and J.-J. Meyer. Agent programming with declarative goals. In C. Castelfranchi and Y. LespÃl'rance, editors, *Intelligent Agents VII (Agent Theories Architectures and Languages)*, volume 1986 of

Lecture Notes in Computer Science, pages 228–243. Springer Berlin Heidelberg, 2001.

[48] K. V. Hindriks. Programming rational agents in GOAL. In A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, editors, *Multi-Agent Programming: Languages, Tools and Applications*, pages 119–157. Springer US, 2009.

[49] I. Horrocks. The FaCT system. In H. de Swart, editor, *Proc. of the 2nd Int. Conf. on Analytic Tableaux and Related Methods (TABLEAUX'98)*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 307–312. Springer, 1998.

[50] I. Horrocks and A. Voronkov. Reasoning support for expressive ontology languages using a theorem prover. In *Proceedings of the 4th International Symposium on Foundations of Information and Knowledge Systems (FoIKS'06)*, pages 201–218, 2006.

[51] Z. Huang, F. van Harmelen, and A. ten Teije. Reasoning with inconsistent ontologies. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 454–459, 2005.

[52] M. Huhns and D. Bridgeland. Multiagent truth maintenance. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1437–1445, 1991.

[53] JIProlog. `http://www.ugosweb.com/jiprolog/`, 2011.

[54] L. Kagal, C. Hanson, and D. Weitzner. Using dependency tracking to provide explanations for policy management. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, POLICY '08, pages 54–61, Washington, DC, USA, 2008. IEEE Computer Society.

[55] A. Kalyanpur. *Debugging and repair of OWL ontologies*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2006. Adviser-Hendler, James.

[56] A. Kalyanpur, B. Parsia, M. Horridge, and E. Sirin. Finding all justifications of OWL DL entailments. *The Semantic Web*, pages 267–280, 2008.

[57] A. Kalyanpur, B. Parsia, E. Sirin, and J. Hendler. Debugging unsatisfiable classes in OWL ontologies. *Journal of Web Semantics*, 3(4):268–293, 2005.

[58] G. Kelleher and L. van der Gaag. The LazyRMS: Avoiding work in the ATMS. *Computational Intelligence*, 9(3):239–253, 1993.

[59] J. S. C. Lam, D. H. Sleeman, J. Z. Pan, and W. W. Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. *Journal of Data Semantics*, 10:62–95, 2008.

[60] S. C. Lam, J. Z. Pan, D. Sleeman, and W. Vasconcelos. A fine-grained approach to resolving unsatisfiable ontologies. In *WI '06: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 428–434, Washington, DC, USA, 2006. IEEE Computer Society.

[61] B. Malheiro, N. R. Jennings, and E. Oliveira. Belief revision in multi-agent systems. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 294–298, 1994.

[62] J. P. Martins and S. C. Shapiro. A model for belief revision. *Artificial Intelligence*, 35, 1988.

[63] M. Marx. Tolerance logic. *Journal of Logic, Language and Information*, 10(3):353–374, 2001.

[64] T. A. Meyer, K. Lee, R. Booth, and J. Z. Pan. Finding maximally satisfiable terminologies for the description logic ALC. In *Proceedings of The 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (AAAI'06)*. AAAI Press, 2006.

[65] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, 1990.

[66] H. Nguyen, N. Alechina, and B. Logan. Ontology debugging with truth maintenance systems. In A. Bundy, J. Lehmann, G. Qi, and I. J. Varzinczak, editors, *ECAI-10 Workshop on Automated Reasoning about Context and Ontology Evolution (ARCOE-10), Workshop Notes*, pages 13–14, Lisbon, Portugal, August 2010.

[67] H. Nguyen, N. Alechina, and B. Logan. Axiom pinpointing using an assumption-based truth maintenance system. In Y. Kazakov, D. Lembo, and F. Wolter, editors, *Proceedings of the 25th International Workshop on Description Logics (DL 2012)*, pages 290–300, Rome, Italy, June 2012. CEUR Workshop Proceedings Vol-846.

[68] H. H. Nguyen. Belief revision in a fact-rule agent's belief base. In *Proceedings of the Third KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, KES-AMSTA '09, pages 120–130, Berlin, Heidelberg, 2009. Springer-Verlag.

[69] I. Niles and A. Pease. Towards a standard upper ontology. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS'01)*, pages 2–9, New York, NY, USA, 2001. ACM.

[70] A. Pease. The Sigma ontology development environment. In *Working Notes of the IJCAI 2003 Workshop on Ontology and Distributed Systems*, volume 71, 2003.

[71] A. Pease. Standard Upper Ontology Knowledge Interchange Format. `http://sigmakee.cvs.sourceforge.net/*checkout*/sigmakee/sigma/suo-kif.pdf`, 2009.

[72] A. Pease and C. Benzmüller. Sigma: An integrated development environment for logical theory development. In *The ECAI 2010 Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET'10)*, Lisbon, Portugal, 2010.

[73] R. D. F. (RDF). `http://www.w3.org/RDF/`, 2011.

[74] R. Reiter. On closed world data bases. pages 55–76, 1977.

[75] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[76] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of the 6th National Conference on Artificial Intelligence, AAAI'87*, pages 183–189, 1987.

[77] Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM 2011)*, 2011.

[78] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.

[79] S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., 2010.

[80] S. Schlobach. Diagnosing terminologies. In *Proceedings of the 20th National Conference on Artificial intelligence (AAAI'05)*, pages 670–675. AAAI Press, 2005.

[81] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In G. Gottlob and T. Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 355–360. Morgan Kaufmann, August 2003.

[82] S. Schlobach, Z. Huang, R. Cornet, and F. van Harmelen. Debugging incoherent terminologies. *Journal of Automated Reasoning*, 39(3):317–349, 2007.

[83] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1 – 26, 1991.

[84] S. Schulz. E - a brainiac theorem prover. *Journal of AI Communications*, 15(2-3):111–126, 2002.

[85] S. C. Shapiro. Belief revision and truth maintenance systems: An overview and a proposal. Technical report, SUNY-Buffalo, 1998.

[86] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5(2):51–53, June 2007.

[87] L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.

[88] SWI-Prolog. `http://www.swi-prolog.org/`, 2011.

[89] T. Swift and D. S. Warren. Xsb: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12:157–187, 1 2012.

[90] A.-Y. Turhan. Reasoning and explanation in $\mathcal{EL}$ and in expressive description logics. In U. Aẅmann, A. Bartho, and C. Wende, editors, *Reasoning Web*, number 6325 in LNCS, pages 1–27. Springer, 2010.

[91] M. Winikoff. Jack$^{\mathrm{tm}}$ intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. E. Fallah-Seghrouchni, editors, *Multi-Agent Programming*, pages 175–193. 2005.

[92] T. Winograd. Procedures as a representation for data in a computer program for understanding natural language. *Cognitive Psychology*, 3(1):1–191, 1971.

[93] G. Wu, G. Qi, and J. Du. Finding all justifications of OWL entailments using TMS and MapReduce. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 1425–1434, New York, NY, USA, 2011. ACM.