ABSTRACT

Title of dissertation: COMBINING DESCRIPTION LOGIC REASONING WITH AI PLANNING FOR COMPOSITION OF WEB SERVICES

Evren Sirin, Doctor of Philosophy, 2006

Dissertation directed by: James Hendler
Department of Computer Science

As Web Services become more prevalent — with the aim of achieving interoperability between heterogeneous, decentralized and distributed systems — the problem of selecting and composing services to accomplish a given task becomes more important. Using Web ontologies to describe different properties of Web Services provided by separate developers facilitates their integration. Automating the composition of Web Services is essential for various different subjects ranging from ordinary users performing tasks on the Web, businesses carrying out complex transactions, and scientists collaborating with each other on the computational Grid.

In this thesis I present the HTN-DL formalism which combines Hierarchical Task Network (HTN) planning and Description Logics (DL) to automatically compose Web Services which are described with Web Ontology Language (OWL).

The main contributions of this thesis are as follows:

- The HTN-DL formalism, which couples Hierarchical Task Network (HTN) planning and Description Logics. HTN-DL combines the expressivity of De-

scription Logics with the efficiency of HTN planning systems to solve Web Service composition problems.

- A translation algorithm from the Semantic Web Service language OWL-S to HTN-DL. This translation algorithm shows that the control constructs used to describe the control flow of a Web Service workflow can be encoded in an HTN-DL domain. The translation also provides a semantics for OWL-S processes; and it is also shown that this semantics is compatible with the previously proposed Situation Calculusbased semantics of OWL-S.

- Novel optimization techniques for DL reasoning which target nominals and large number of individuals. These optimization techniques allow the HTN-DL planner to efficiently reason with OWL-DL ontologies during planning. Our empirical analysis shows that these optimizations dramatically improve consistency checking, classification, and realization tasks.

- Optimizations for conjunctive query answering w.r.t. DL knowledge bases. Inspired by some query optimization techniques used in relational databases, a cost-based model is presented to estimate the evaluation time of DL queries.

- An implementation of the HTN-DL planning system that interacts directly with Web Services. The components of the planning system, OWL-DL reasoner Pellet and API for OWL-S services, are also released as stand-alone tools and have been incorporated into many systems.

COMBINING DESCRIPTION LOGIC REASONING WITH AI PLANNING FOR COMPOSITION OF WEB SERVICES

by

Evren Sirin

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Commmittee:
Professor James Hendler, Chair
Professor Ashok Agrawala
Professor Mark Austin
Professor Ian Horrocks
Dr. Ryusuke Masuoka
Professor Dana Nau

© Copyright by

Evren Sirin

2006

DEDICATION

To my family

ACKNOWLEDGMENTS

I wish to thank a number of people who have supported, directed, and assisted me in completing this thesis. First of all, I would like to thank my advisor, James Hendler, whose support, advice and encouragement throughout the years helped me tremendously. Under the roof of MINDSWAP, he created a research environment full of enthusiasm and inspiration which made this thesis possible.

Out of the MINDSWAP members, above all, I would like to express my heartfelt gratitude for Bijan Parsia who helped me in every aspect of my research. His guidance and inspiration helped me to start my thesis and his continuous nagging helped me to finish it. I would also like to thank my colleagues Bernardo Cuenca Grau and Aditya Kalyanpur for fruitful discussions and the papers we co-authored some of which contributed to my thesis. I have received a lot of support from other members of the MINDSWAP group including Ron and Amy Alford, Jennifer Golbeck, Christian Halaschek, Yarden Katz, Vladimir Kolovski, David Taowei Wang, and others I may have forgotten. Special thanks to Kendall Clark for his careful proofreading.

I would also like to express how valuable it was for me to collaborate with many knowledgeable and helpful people along the way, Prof. Dana Nau and two of his students, Dan Wu and Ugur Kuter; Ryusuke Masuoka and many other people from the Fujitsu Labs of America, College Park; the members of the OWL-S coalition, just

to name a few. Ian Horrocks and his detailed comments about the thesis improved the quality of the document a lot.

It is important to note that life at College Park would be much harder without my dear friends, Fazil, Burcu, Tikir, Okan and others. I owe so much to my parents Serife and Mehmet Sirin and my brother Uygar Sirin who have been the best role models for me and always supported me with their endless love. I am especially grateful to my wife Fusun Yaman who always provided help and comfort even though she had her own thesis to finish. I would not have gone through the painful experience of getting a PhD if it wasn't for her love and company. I also would like thank my daughter Gizem whose expected arrival was the best motivation for me to finish my thesis.

TABLE OF CONTENTS

List of Algorithms ix

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Motivation 1

1.2 Proposed Solution 7

1.3 Contributions 8

1.4 Thesis Outline 9

2 Preliminaries 11

2.1 Web Services 11

2.2 Semantic Web 13

2.3 Description Logics 14

2.3.1 Syntax 15

2.3.2 Semantics 17

2.3.3 Inference Problems 18

2.3.4 A Tableau Algorithm for $\mathcal{SHOIN}$ 23

2.4 Semantic Web Services 28

2.5 AI Planning 29

3 Coupling Planning with Description Logics: HTN-DL 33
3.1 AI Planning and Web Service Composition 33
3.2 HTN-DL 37
3.2.1 Overview 37
3.2.2 Syntax 43
3.2.3 Semantics 55
3.3 HTN-DL Algorithm 64
3.3.1 Evaluating Conditions 65
3.3.2 Updating State 69
3.3.3 Interleaving Execution and Planning 72
4 Translating Web Service Descriptions to HTN-DL 75
4.1 Relation between OWL-S and HTN-DL 76
4.2 From OWL-S to HTN-DL 79
4.2.1 Translating Profile Descriptions 79
4.2.2 Translating Process Models 80
4.3 OWL-S semantics 93
5 Optimizing OWL-DL Reasoning 99
5.1 Reasoning with Nominals in OWL-DL 101
5.2 Preprocessing Optimizations 105
5.2.1 Existing Optimizations 105
5.2.2 Nominal Absorption 107
5.3 Optimizations for Consistency Checking 111

5.3.1 Existing Optimizations 113

5.3.2 Learning-based Disjunct Selection 114

5.3.3 Completion Graph Caching 116

5.3.4 Lazy Completion Graph Generation 118

5.4 Optimizations for Subsumption and Instance Checking 119

5.4.1 Nominal-based Model Merging 120

6 Efficient Conjunctive Query Answering 126

6.1 Answering Atomic Queries 127

6.1.1 Retrieving Instances 128

6.1.2 Retrieving Role Fillers 131

6.2 Answering Conjunctive Boolean Queries 133

6.3 Answering Conjunctive Retrieval Queries 135

6.4 Cost-based Query Reordering 136

6.4.1 Size and Cost Estimation 141

6.5 Query Simplification 142

7 Implementation and Evaluation 144

7.1 System Architecture 144

7.2 Pellet: OWL-DL Reasoner 146

7.2.1 Pellet Architecture and Design 146

7.2.2 Tableaux Reasoner 147

7.2.3 OWL Species Coercion 150

7.2.4 ABox Query Engine 151

7.2.5 Special Features . . . 153
7.3 OWL-S API: API for Web Service . . . 155
7.3.1 The Design Objectives . . . 156
7.3.2 Architecture of the OWL-S API . . . 157
7.4 HTN-DL: Planning for Web Services . . . 159
7.5 Experimental Evaluation . . . 159
7.5.1 Reasoning Performance . . . 160
7.5.2 Planning Performance . . . 167

8 Related Work 173
8.1 Description Logics and Planning . . . 173
8.2 Description Logics and Web Services . . . 177
8.3 Web Service Composition and Planning . . . 180

9 Conclusions 184
9.1 Summary . . . 184
9.2 Contributions and Impact . . . 185
9.3 Discussion . . . 187
9.4 Future Work . . . 189

A Proofs 193

Bibliography 208

LIST OF ALGORITHMS

1 HTN-DL planning algorithm 65

2 Finding applicable methods and operators 66

3 Translation of a generic OWL-S constrol construct 84

4 Translation of a `Sequence` construct 87

5 Translation of a `Choice` construct 88

6 Translation of a `Split-Join` construct 89

7 Translation of an `If-Then-Else` construct 90

8 Translation of a `Repeat-While` construct 91

9 Translation of a `Repeat-Until` construct 98

10 Modified absorption algorithm 112

11 Learning-based disjunct selection algorithm 117

12 Finding obvious instances and non-instances 132

13 Conjunctive query answering algorithm for retrieval queries 137

14 Query cost estimation for conjunctive retrieval queries 140

LIST OF TABLES

2.1 Semantics of $\mathcal{SHOIN}$ axioms 18

2.2 Tableau expansion rules for $\mathcal{SHOIN}$ 27

3.1 Task matching based on preconditions and effects 42

4.1 Golog constructs used to represent OWL-S processes 94

7.1 Evaluation of optimization strategies implemented in Pellet 161

7.2 Evaluating size estimation performance and accuracy with respect to sampling ratio. 165

7.3 Comparison of query evaluation times for different orderings 167

LIST OF FIGURES

3.1 A composite Web Service example using a task ontology 40

5.1 Completion graphs for concepts *RedWine* and *ItalianWine* 121

5.2 Completion graphs for concepts *DryWine* and *NonSweetWine* 123

7.1 Overview of HTN-DL planning system 145

7.2 Main components of the Pellet reasoner 147

7.3 Different completion strategies implemented in Pellet 148

7.4 Components of the query engine 152

7.5 Basic components of the OWL-S API 158

7.6 Correlation between the cost estimates and the query answering time 166

7.7 Comparison of HTN-DL with JSHOP system 170

7.8 Performance of HTN-DL planner 172

Chapter 1

Introduction

1.1 Motivation

The World Wide Web has become a part of daily life. Many different tasks — shopping, financial transactions, and travel arrangements — are now being accomplished as a routine matter of daily life over Web. When users interact with Web sites to accomplish these tasks, there are a number of issues they need to solve. Finding the right Web site to provide the requisite service is a challenge as generally there are many possibilities. Often there is no one such provider and the user needs to visit several different Web sites to accomplish the overall goal.

Let us consider some scenarios which illustrate users accomplishing some tasks on the Web:

1. Bob is feeling sick with several symptoms. He goes to an online health service that suggests an over-the-counter medicine for his symptoms. He then finds a pharmacy Web site to locate a pharmacy that sells the medicine and is near to his location. Then he gets the directions to the pharmacy using a map service.

2. Bob's symptoms have not improved, so he decides to see a specialist for an examination. He checks the Web service of his health insurance provider to find the approved specialists in the area. He chooses a doctor and contacts

the doctor's Web site to make an appointment. He records the appointment time in his personal calendar and sets up a reminder to notify him one day before the appointment.

3. Bob lives in Washington, DC and is making travel arrangements to attend a conference in Kyoto, Japan. He first needs to register for the conference using the conference Web site. To make his flight arrangements, he can use three airports near Washington, DC and two airports near Kyoto. He also needs to make local transportation arrangements. Depending on the airport selection, he may choose to drive (his own car in DC or a rental car in Japan) or use public transportation for getting to and from the airports. To make his accommodation arrangements, he looks for hotels near the conference venue. He finds a hotel that has an available room for the dates he is traveling. After everything is finalized, he sends the final travel itinerary to the business office to initiate the reimbursement process for his expenses. Finally, he publishes the dates he will be traveling to an online shared calendar hosted by his research lab so that his colleagues will be informed of his schedule.

4. Bob wants to get the DVDs of the 6 movies in the Star Wars series. He does not have a preference between getting all of them at once or geting them separately, though he would prefer to get the special edition set that contains episodes IV, V, and VI together. In either case, he wants the DVDs of those episodes to be new and unused, but he can get the other episodes second-hand to reduce the total cost.

Bob faces several different problems in each scenario. One problem is to find the best service, according to his definition of best, that provides the functionality he desires. For example, he might need to go through a number of different pharmacy Web sites to find the closest one or look at different DVD stores to find the cheapest price. While looking at the DVD stores, he needs to verify which ones are selling used DVDs and which ones are selling new ones. In the case of his trip to Kyoto, there might be incompatibilities with how different services work. For example, the hotel cost might be billed in Japanese yen, but the all the items in the reimbursement request should be reported in US dollars. Resolving all these issues manually is time-consuming, error-prone, and thankless.

The aim of the Web Services paradigm [12] is to provide a standard description language for the services available on the Web. The idea is to describe the functionality of services in a machine interpretable way so that interaction with these services can be achieved without human control. Such descriptions typically define the structure of messages that need to be sent in order to execute a service and the format of the output that will be returned by the service. However, since the Web is a distributed and uncontrolled environment, describing syntactic features of Web Services is not sufficient: different Web Service developers might use different structures for their services. For example, when Bob is looking for a pharmacy , the address returned by the pharmacy locator might be structures with several fields such as street address, city, state, etc. However, the map service might require the addresses as one concatenated string. In this case, even though the message contents used by both services are semantically the same, the automatic execution of

the composition would fail if the conversion between message types is not done.

Providing more semantics for Web Service descriptions can be achieved using ontologies on the Semantic Web. The Semantic Web vision [9, 51] is of a world where loosely coupled, independently evolving ontologies provide common understandings between heterogeneous agents, systems, and organizations. Such ontologies can be exploited to provide fairly rich descriptions for several different aspects of Web Services, including categorization of services, semantics of the message contents, information about the provider, and so on.

Given such Web Service descriptions, the main steps involved in the preceding scenarios about Bob can be summarized thus: *discovering* the available services on the Web; *matching* the capabilities of the services with the user's objective; *composing* multiple services to fulfill all the requirements; and, finally, *executing* the generated composition automatically.

The main motivation of this thesis is the problem of composing Semantic Web Services. Matching service capabilities is very closely related and cannot be considered separately. Discovery[1] and execution are considered as separate issues.

Although the scenarios we have mentioned above only concern ordinary Web users, similar problems come up in other settings. For example, consider the growing multidisciplinarity of many large-scale scientific reasearch projects. Scientific collaborations on the Grid infrastructure are increasing and moving towards service-based architectures [31]. Many tasks on the Grid require the coordination and combination of multiple services and resources. Composing these services in workflows of

[1]By which term we mean *locating Web Service description files*

varying complexity is required for different tasks [10, 2]. For example, a Grid application might retrieve data from one source, use different programs to analyze and transform the data, combine these results in a particular format, and then send it to another service for further processing.

Similar scenarios also occur in B2B applications [8, 18, 99]. The composition of services is very interesting from a business perspective because online partnerships can automatically be formed without prior agreements. A business that wants to order some items from a manufacturer and then arrange the shipment details can achieve this goal by combining the services provided by manufacturers and shipment companies. As in previous examples, the dynamic composition of services in this context also requires understanding of the service capabilities and the compatibility between available services.

Some fundamental characteristics shared by all these examples include the following:

- *Decentralized Setting* Service descriptions are created by different providers that do not necessarily use the same vocabulary or structure to describe the services.

- *Service Attributes* There are many Web Services with similar functionality and capabilities. The only distinguishing features between services might be attributes such as who is providing the service, what kind of credentials they have, what kind of security policies are being used in the communication. These attributes need to be taken into consideration while deciding whether

a service will be included in the composition or not.

- *Composite Services* Not all Web Services are atomic, some are composite services that are constructed from other (possibly composite) services. For example, a composite service might describe several different steps of interacting with a Web site. Alternatively, composite services can act as reusable, customizable templates outlining the standard operating procedures for accomplishing a task. Therefore, it is important to describe and reason with such composite service descriptions.

- *Open World* For Web Service composition problems, most typically we do not have complete information about the world. It is not realistic to adopt the closed world assumption and assume that what we do not know is false. When reasoning about the world, we need to take into consideration that we have incomplete information.

- *Interleaved Execution and Composition* The composition system might not have all the relevant information to solve a composition problem but some of the Web Services provide information that is relevant. Such information-providing services should be executed during composition so that information gathered can be used to build the composition.

- *Efficiency* There are many services available on the Web. During composition, reasoning about these services, e.g. matching their capabilities and considering how the execution of one service will affect other services, should be done very

efficiently.

1.2 Proposed Solution

This thesis presents HTN-DL which combines HTN planning formalism with Description Logic representation. There are many novel features of HTN-DL that makes it suitable for solving Web Service composition problems. An expressive knowledge representation language with Open World semantics is used to represent the state of the world. Actions and tasks are also described using an ontology so that task matching can be done in a flexible way. HTN-DL also differentiates between world-altering effects and knowledge effects making it possible to execute information-providing services during compositions.

A forward-decomposition planning algorithm is presented to solve HTN-DL problems. The algorithm takes into consideration that we have incomplete knowledge and therefore the truth value of a condition can take three values: `true`, `false`, and `unknown`. Only plans that are guaranteed to be sound with the given knowledge are found. Services that have only knowledge effects are executed during composition.

It is shown that several control structures that are commonly used to model composite Web Services can be expressed as HTN-DL methods. Specifically, an algorithm for translating process models expressed in the Semantic Web Service language OWL-S to HTN-DL is given. The correctness of the compositions generated from the resulting planning domain is shown with respect to the Situation Calculus

semantics of OWL-S.

As the planning system relies on the inferences drawn by the DL reasoner, the practicality of the proposed solution crucially depends on the efficiency of the DL reasoner. For this reason, several novel optimization techniques, especially geared toward handling nominals and large number of individuals, are presented. The empirical analysis shows that these optimizations can dramatically improve consistency checking, classification, and realization tasks. The reasoning service that is frequently used by the the HTN-DL planning system is conjunctive query answering. Optimization techniques for conjunctive query answering inspired by the techniques used in relational databases are presented in order to improve query evaluation times.

1.3 Contributions

The contributions of this thesis are as follows:

- The HTN-DL formalism, which couples HTN planning and Description Logics, combines the expressivity of Description Logics with the efficiency of HTN planning systems to solve Web Service composition problems. The hierarchical structure of HTN-DL domains can conveniently describe composite Web Service descriptions and fit in well with the loosely coupled nature of Web Services. Ontology-based reasoning provides a flexible mechanism to reuse the Web Services that are defined by separate developers in different contexts.

- A translation algorithm from OWL-S to HTN-DL is provided in order to show

that the control constructs used to describe the control flow of a Web Service workflow can be encoded as HTN-DL domains. The translation provides a semantics for OWL-S processes and is shown to be compatible with the previously proposed Situation Calculusbased semantics of OWL-S.

- Novel optimizations for DL reasoning targeting nominals and large number of individuals are presented. Our empirical analysis shows that these optimizations drastically improve consistency checking, classification, and realization tasks.

- Optimizations for conjunctive query answering w.r.t. DL knowledge bases are introduced. Inspired by query optimization techniques used in relational databases, a cost-based model is presented to estimate the evaluation time of DL queries. We propose efficient heuristics to compute the costs of queries and demonstrate the effectiveness of the query optimization techniques empirically.

- An implementation of HTN-DL planning system that interacts directly with Web Services is presented. The components of the planning system, the OWL-DL reasoner Pellet and the API for OWL-S services, are also released as standalone tools and have been incorporated in many systems.

1.4 Thesis Outline

This thesis is organized as follows:

- Chapter 2 presents briefly the background information required to follow the

theory in this thesis.

- Chapter 3 discusses how AI planning techniques can be used for Web Service composition problems. We examine the common issues and problems that arise when we try to model Web Services in a planning domain. Then we present HTN-DL which combines HTN planning formalism with Description Logic representation to overcome these problems.

- Chapter 4 presents a translation algorithm that generates HTN-DL domains from OWL-S descriptions.

- Chapter 5 explains optimization techniques designed for the DL $\mathcal{SHOIN}$, the DL underlying OWL-DL. We focus on methods that can improve the efficiency of standard reasoning services such as KB consistency, classification, and realization.

- Chapter 6 presents optimization methods to improve conjunctive query answering which greatly effects the performance of HTN-DL.

- Chapter 7 describes the system architecture of the HTN-DL system and its components namely; Pellet, an OWL-DL reasoner, the OWL-S API, an API designed for Web Services. We also present the empirical evaluation of the system performance.

- Chapter 8 discusses the related work.

- Finally Chapter 9 concludes with the summary, impact of the thesis and discussions about the future work.

Chapter 2

Preliminaries

In this chapter, we provide some background on Web Services, Semantic Web, Description Logics, and AI planning. The purpose of this chapter is to describe the basic concepts, introduce the necessary terminology, and present relevant definitions.

2.1 Web Services

There are various different standards that have been developed for different Web Service tasks such as description, discovery and invocation. These technologies are primarily designed to be used in conjunction with other Web standards, e.g. XML for syntax and HTTP for communication.

SOAP [45] is the communication protocol designed to exchange messages between applications over the Web. It is fundamentally a stateless, one-way message exchange paradigm, but applications can create more complex interaction patterns by combining such one-way exchanges. SOAP provides a distributed processing model where a SOAP message is delivered from a sender to an ultimate receiver via zero or more SOAP intermediaries. This distributed processing model can support many message exchange patterns including but not limited to one-way messages, request/response interactions, and peer-to-peer conversations.

Web Service Description Language (WSDL) [19] is the language to describe the

mechanics of interacting with a particular Web service. The abstract functionality of the Web service is defined in terms of the types of messages it sends and receives in WSDL *interface*. An interface is a set of *operations* and an operation is a sequence of input and output messages. An operation associates a message exchange pattern (MEP) with the message types that will be exchanged during execution. The message types are defined using a schema language such as (but not limited to) XML Schema. The abstract interfaces are associated to concrete message formats and transmission protocols with *binding* descriptions.

Universal Description Discovery and Integration (UDDI) [114] is an emerging standard registry system for Web Services. UDDI allows businesses to advertise their Web Services by publishing their descriptions on a global registry. There are three main parts of this registry: White Pages that list contact information about the company that developed the Web service; Yellow Pages that organize Web services by such categories as geography and industry code; and Green Pages that hold WSDL descriptions. UDDI supports the association of an unbounded set of properties to the description of Web Services via a construct called TModel. For example, a service may specify its category using an arbitrary classification system though their meaning is not codified, therefore there may be two different TModels with the same meaning, but this similarity cannot be recognized.

2.2 Semantic Web

The Semantic Web [9] is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. This is realized by marking up Web content, its properties, and its relations, in a reasonably expressive markup language with a well-defined semantics.

Semantic Web languages are used to represent information about resources on the Web. This information is not limited to be about Web resources but can be about anything that can be identified. Uniform Resource Identifiers (URIs) are used to uniquely identify entities. For example, it is possible to assign a URI to a person, to the company he works for, to the car he owns, etc. so relations between these entities can be written and shared on the Semantic Web.

There is a stack of languages that have been published as W3C recommendations to be used on Semantic Web. At the bottom layer of the stack, there is the Resource Description Framework (RDF) [16]. RDF is a simple assertional language that is designed to represent information in the form of triples. Triples are statements that contain a subject, a predicate and an object. RDF Schema (RDFS) [15] is a collection of RDF resources that can be used to describe properties of other RDF resources. Unlike its name suggests, RDFS is not a schema that imposes specific constraints on the structure of a document, but instead it provides information about the interpretation of the statements given in an RDF data model. In this regard, RDFS has similarities to frame based languages and can even be described as a relatively inexpressive Description Logic (DL).

The Web Ontology Language (OWL) [23], is the most expressive standardized Semantic Web language that is layered on top of RDF and RDFS. OWL can be used to define *classes* (unary relations) and *properties* (binary relations) as in RDFS but also provides constructs to create new class descriptions as logical combinations (intersections, unions, or complements) of other classes, define cardinality restrictions on properties and so on. OWL has three different species: OWL-Lite, OWL-DL and OWL-Full. OWL-Lite and OWL-DL differ from OWL-Full such that they define certain constraints on RDF and RDFS so as to be compatible with the traditional semantics of DLs.

The semantics of unrestricted RDF-S and OWL-Full is non-traditional and the reasoners built for OWL Full fragment tend to be sound but incomplete. Since there is no straight-forward way to extend the existing reasoners to support the full expressivity of OWL-Full. Therefore, we will be focusing on OWL-DL fragment of the language and use sound and complete reasoning techniques developed for Description Logics.

2.3 Description Logics

Description Logics are a family of class-based knowledge representation formalisms [3]. A DL knowledge base typically comprises two components: a “TBox” and an “ABox”. The TBox contains intensional knowledge in the form of a terminology and the ABox contains extensional knowledge that is specific to the individuals of the domain of discourse. Intensional knowledge is usually thought to change

rarely and extensional knowledge is usually thought to be contingent, or dependent on a single set of circumstances, and therefore subject to occasional or even constant change [3].

In the rest of the section, we briefly describe the syntax and semantics of the Description Logic $\mathcal{SHOIN}$ which is the DL underlying OWL-DL.

2.3.1 Syntax

Let N_C, N_R, N_I be non-empty and pair-wise disjoint sets of *atomic concepts*, *atomic roles*, and *individuals* respectively. The set of $\mathcal{SHOIN}$ roles is the set $N_R \cup \{R^- \mid a \in N_R\}$, where R^- denotes the inverse of the atomic role R. To avoid considering roles such as R^{--}, we define the function Inv such that $\mathsf{Inv}(R) = R^-$ and $\mathsf{Inv}(R^-) = R$ for $R \in N_R$.

A *role inclusion axiom* is an expression of the form $R \sqsubseteq S$, where $R, S \in N_R$. A *transitivity axiom* is an expression of the form $\mathsf{Trans}(R)$, where $R \in N_R$. An RBox $\mathcal{R}$ is a finite set of role inclusion axioms and transitivity axioms.

Let $\sqsubseteq^*$ be the reflexive-transitive closure of $\sqsubseteq$. A role R is transitive if there is a role S such that $\mathsf{Trans}(S) \in \mathcal{R}$ with $S \sqsubseteq^* R$ and $R \sqsubseteq^* S$. R is *simple* if there is no role S such that $S \sqsubseteq^* R$ and S is transitive. R is *complex* if it is not simple.

The set of $\mathcal{SHOIN}$-concepts (or concepts for short) is defined inductively as the minimal set for which the following holds:

1. every concept name $C \in N_C$ is a concept,

2. if C and D are concepts and R is a role, then $(C \sqcap D)$ (conjunction), $(C \sqcup D)$

(disjunction), $(\neg C)$ (negation), $(\exists R.C)$ (existential restriction), and $(\forall R.C)$ (value restriction) are also concepts, and

3. if C is a concept, R is a simple role and $n \in \mathbb{N}$, then $(\leq nR)$ and $(\geq nR)$ are also concepts (called at-most and at-least number restrictions), and

4. if $a \in N_I$ is an individual, then $\{a\}$ is also a concept.

We use $\top$ and $\bot$ to abbreviate $A \sqcup \neg A$ and $A \sqcap \neg A$, respectively, where A is a concept name.

For C, D concepts, a *concept inclusion axiom* is an expression of the form $C \sqsubseteq D$. A TBox $\mathcal{T}$ is a finite set of concept inclusion axioms. A *concept equivalence axiom* has the form $C \equiv D$ and is simply an abbreviation for $C \sqsubseteq D$ and $D \sqsubseteq C$. Axiom $C \sqsubseteq D$ is called a *primitive definition* if C is a concept name and it is called a *general concept inclusion* (GCI) if C is a complex concept. A TBox $\mathcal{T}$ is called a *general TBox* if it contains GCIs.

An ABox $\mathcal{A}$ is a finite set of concept and role membership axioms of the form $C(a)$, $R(a, b)$, $\neg R(a, b)$, and (in)equality axioms $a \approx b$ and $a \not\approx b$, where C is a $\mathcal{SHOIN}$ concept, R is a role, and a and b are individuals.

A *knowledge base* $\mathcal{K}$ is a triple $(\mathcal{A}, \mathcal{T}, \mathcal{R})$ where $\mathcal{A}$ is an ABox, $\mathcal{T}$ is a TBox and $\mathcal{R}$ is an RBox.

To improve readability, we will use the shorthand notation $\mathcal{K} \cup \{\alpha\}$ to denote the addition of the axiom α to one of the ABox or the TBox component of the KB $\mathcal{K}$. Clearly, if α is an ABox (resp. TBox) axiom it will be added to the ABox (resp. TBox) component. More formally, if $\mathcal{K} = (\mathcal{A}, \mathcal{T}, \mathcal{R})$ and $\mathcal{K}' = \mathcal{K} \cup \{C(a)\}$ then

$K' = (\mathcal{A}', \mathcal{T}, \mathcal{R})$ where $\mathcal{A}' = \mathcal{A} \cup \{C(a)\}$. Similarly, if $\mathcal{K}' = \mathcal{K} \cup \{C \sqsubseteq D\}$ then $K' = (\mathcal{A}, \mathcal{T}', \mathcal{R})$ where $\mathcal{T}' = \mathcal{T} \cup \{C \sqsubseteq D\}$.

2.3.2 Semantics

An *interpretation* $\mathcal{I}$ is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty set, called the *domain* of the interpretation, and $\cdot^{\mathcal{I}}$ is the *interpretation function*. The interpretation function maps each atomic concept $A \in N_C$ to a subset of $\Delta^{\mathcal{I}}$, each atomic role $R \in N_R$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, each individual $a \in N_I$ to an element of $\Delta^{\mathcal{I}}$. The interpretation function is extended to concept descriptions as follows:

- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\exists R.C)^{\mathcal{I}} = \{x \mid \exists y : (x, y) \in R^{\mathcal{I}} \land y \in C^{\mathcal{I}}\}$
- $(\forall R.C)^{\mathcal{I}} = \{x \mid \forall y : (x, y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
- $(\geq nR)^{\mathcal{I}} = \{x \mid \sharp\{y \mid (x, y) \in R^{\mathcal{I}}\} \geq n\}$
- $(\leq nR)^{\mathcal{I}} = \{x \mid \sharp\{y \mid (x, y) \in R^{\mathcal{I}}\} \leq n\}$
- $\{a\}^{\mathcal{I}} = \{a^{\mathcal{I}}\}$

The interpretation function is extended to complex roles as follows:

- $(\mathsf{Inv}(R))^{\mathcal{I}} = \{(x, y) \mid (y, x) \in R^{\mathcal{I}}\}$

Axiom type	Syntax	Satisfaction Condition
Concept inclusion	$C \sqsubseteq D$	$C^\mathcal{I} \subseteq D^\mathcal{I}$
Role inclusion	$R \sqsubseteq S$	$R^\mathcal{I} \subseteq S^\mathcal{I}$
Transitivity	$\mathsf{Trans}(R)$	$(R^\mathcal{I})^+ \subseteq R^\mathcal{I}$
Concept membership	$C(a)$	$a^\mathcal{I} \in C^\mathcal{I}$
Role membership	$R(a,b)$	$(a^\mathcal{I}, b^\mathcal{I}) \in R^\mathcal{I}$
Negated role membership	$\neg R(a,b)$	$(a^\mathcal{I}, b^\mathcal{I}) \notin R^\mathcal{I}$
Equality	$a \approx b$	$a^\mathcal{I} = b^\mathcal{I}$
Inequality	$a \not\approx b$	$a^\mathcal{I} \neq b^\mathcal{I}$

Table 2.1: Semantics of $\mathcal{SHOIN}$ axioms

The satisfaction of a $\mathcal{SHOIN}$ axiom in an interpretation $\mathcal{I}$ is given by the conditions listed in Table 2.1.

If an interpretation $\mathcal{I}$ satisfies an axiom α we say $\mathcal{I} \models \alpha$. The interpretation $\mathcal{I}$ is a model of the ABox $\mathcal{A}$ (resp. TBox $\mathcal{T}$ or RBox $\mathcal{R}$) if it satisfies all the axioms in $\mathcal{A}$ (resp. $\mathcal{T}$ or $\mathcal{R}$). $\mathcal{I}$ is a model of $\mathcal{K} = (\mathcal{A}, \mathcal{T}, \mathcal{R})$, denoted by $\mathcal{I} \models \mathcal{K}$, if $\mathcal{I}$ is a model of $\mathcal{A}$, $\mathcal{T}$ and $\mathcal{R}$.

2.3.3 Inference Problems

The basic inference problem for DL is checking KB consistency. The knowledge base $\mathcal{K}$ is *consistent* if it has a model. The additional inference problems are

- *Concept Satisfiability*, A concept C is *satisfiable relative to* $\mathcal{K}$ if there is a model $\mathcal{I}$ of $\mathcal{K}$ such that $C^\mathcal{I} \neq \emptyset$.
- *Concept Subsumption* A concept C is subsumed by concept D relative to $\mathcal{K}$ if, for every model $\mathcal{I}$ of $\mathcal{K}$, $C^\mathcal{I} \subseteq D^\mathcal{I}$.
- *Concept Instantiation* An individual i is an instance of concept C relative to $\mathcal{K}$ if, for every model $\mathcal{I}$ of $\mathcal{K}$, $a^\mathcal{I} \in C^\mathcal{I}$.

All these reasoning problems can be reduced to KB consistency. For example, concept C is satisfiable w.r.t. the KB $\mathcal{K}$ if $\mathcal{K} \cup \{C(a)\}$ is consistent where a is an individual not occurring in $\mathcal{K}$.

A DL *conjunctive query* $Q(\mathbf{x}, \mathbf{y})$ is a conjunction of terms of the form $C(x)$, $R(x, y)$ or $\neg R(x, y)$, where C is a concept, R is a role, and x, y are individual names taken from N_I or variable names taken from the sets $\mathbf{x}$ and $\mathbf{y}$. The $\mathbf{x}$ and $\mathbf{y}$ are all the variables appearing in the terms and are called distinguished and non-distinguished variables, respectively. A *boolean conjunctive query* is a query that has no distinguished variables, i.e. $\mathbf{x} = \emptyset$. We will use $Q(\mathbf{y})$ to denote boolean queries. A *ground boolean query* is a query that has no variables, i.e. $\mathbf{x} \cup \mathbf{y} = \emptyset$. We will use $Q()$ to denote ground boolean queries. To avoid confusion, we will use the term *retrieval query* for conjunctive queries with distinguished variables. We denote by $VC(Q)$ the set of variables and individuals in Q.

Before defining the semantics for queries let us present some notation that will be useful. A *variable substitution* (or *variable mapping*) is a function from variable names to one of (or a combination of) individual names, elements of the domain, and variable names. The range of the substitution might be different depending on the context and the range will be explicitly specified in the text for each variable substitution. If θ and σ are variable substitutions, *theta*$^-$ denotes the inverse substitution of *theta*, $\theta \cup \sigma$ denotes the union of two substitutions, and $\theta\sigma$ denotes the composition of two substitutions. Note that, the result of these operations might not always be a proper variable substitution, e.g. inverse of a valid variable substitution might be undefined.

Example 2.1 *The following are three valid variable substitutions:*

$$\theta = \{x \rightarrow \mathtt{a}, y \rightarrow \mathtt{a}\}$$

$$\sigma = \{x \rightarrow \mathtt{c}\}$$

$$\rho = \{z \rightarrow x\}$$

Then we have

$$\theta^{-} = \mathit{undefined}$$

$$\rho^{-} = \{x \rightarrow z\}$$

$$\theta \cup \sigma = \mathit{undefined}$$

$$\theta \cup \rho = \{x \rightarrow \mathtt{a}, y \rightarrow \mathtt{a}, z \rightarrow x\}$$

$$\rho\sigma = \{z \rightarrow \mathtt{c}\}$$

$$\rho\theta = \{z \rightarrow x\}$$

If α is a query term and θ is a variable substitution, we use $\alpha\theta$ to denote the term where the variables in α are substituted according to θ. If $Q(\mathbf{x}, \mathbf{y})$ is a DL query then $Q(\mathbf{x}\theta, \mathbf{y})$ denotes the query where all the occurrences of variables in Q has been replaced with the corresponding mapping in θ.

Example 2.2 *Let $Q(\mathbf{x}, \mathbf{y})$ be the following boolean query*

$$Q(\mathbf{x}, \mathbf{y}) = purchased(customer, ticket) \wedge PlaneTicket(ticket)$$
$$\wedge\, NonRefundable(ticket) \wedge livesIn(customer, city)$$

where $\mathbf{x} = \{customer, city\}$ and $\mathbf{y} = \{ticket\}$. Let θ be the following variable substitution.

$$\theta = \{cust \rightarrow \texttt{Bob}, city \rightarrow \texttt{WashingtonDC}\}$$

Applying the substitution θ to the query Q yields the following boolean query

$$Q(\mathbf{x}\theta, \mathbf{y}) = purchased(\texttt{Bob}, ticket) \wedge PlaneTicket(ticket)$$
$$\wedge\, NonRefundable(ticket) \wedge livesIn(\texttt{Bob}, \texttt{WashingtonDC})$$

Now we are ready to describe how to interpret boolean queries. An interpretation $\mathcal{I}$ is a model of a boolean query $Q(\mathbf{y})$, denoted $\mathcal{I} \models \exists \mathbf{y} : Q(\mathbf{y})$ (or shortly $\mathcal{I} \models Q$), if there is a variable substitution $\theta : VC(Q) \rightarrow \Delta^{\mathcal{I}}$ such that $\theta(a) = a^{\mathcal{I}}$ for each individual $a \in VC(Q)$, and $\mathcal{I} \models \alpha\theta$ for each atom α in the query. The notation $\alpha\theta$ denotes the query atom α where the variables of α are substituted according to θ. The knowledge base $\mathcal{K}$ entails a boolean query Q, denoted by $\mathcal{K} \models Q$, if every model of $\mathcal{K}$ satisfies Q.

Now let us consider queries with distinguished variables. An *answer to a query* $Q(\mathbf{x}, \mathbf{y})$ w.r.t. knowledge base $\mathcal{K}$ is a variable substitution θ that maps the distinguished variables of the query $\mathbf{x}$ to the individuals in $\mathcal{K}$ such that the boolean

query $Q(\mathbf{x}\sigma, \mathbf{y})$ is entailed by $\mathcal{K}$ as defined above. Note that, for interpreting boolean queries, we use substitution that maps variables to arbitrary elements of the domain whereas for a query answer we require that the distinguished variables mapped to named individuals in the KB. The *answer set* of a query Q w.r.t. $\mathcal{K}$ (denoted by $Q(\mathcal{K})$) is the set containing all the answers of query Q w.r.t. $\mathcal{K}$. Since a boolean query does not have any distinguished variables, the answer set for a boolean query can either be empty, indicating the boolean query is not entailed by the KB, or it can be a singleton set with one empty mapping, indicating the boolean query is entailed.

A query $Q(\mathbf{x}, \mathbf{y})$ is subsumed by (contained in) query $Q'(\mathbf{x}, \mathbf{y}')$ w.r.t. $\mathcal{K} = \langle \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ (denoted by $\mathcal{K} \models Q \sqsubseteq Q'$), if, for every possible ABox A' and the knowledge base $\mathcal{K}' = \langle \mathcal{A}', \mathcal{T}, \mathcal{R} \rangle$, it holds that $Q(\mathcal{K}') \subseteq Q'(\mathcal{K}')$. Query containment is very closely related to query answering. The standard technique of "query freezing" [115] can be used to reduce query containment problem to query answering in DLs [90]. To decide query subsumption, we build a canonical ABox $\mathcal{A}_Q$ from the query $Q(\mathbf{x}, \mathbf{y})$ by replacing each of the variables in $\mathbf{x}$ and $\mathbf{y}$ with fresh individual names not appearing in the KB. Let θ be the substitution denoting the mapping of variables $\mathbf{x}$ to the fresh individuals. Then, for $\mathcal{K} = \langle \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$, $\mathcal{K} \models Q \sqsubseteq Q'$ iff θ is in the answer set of Q' w.r.t. to $\mathcal{K}_Q = \langle \mathcal{A}_Q, \mathcal{T}, \mathcal{R} \rangle$.

Note that, this standard query subsumption definition is based on the assumption that both Q and Q' share the same set of distinguished variables. In the general case, we say that the query $Q(\mathbf{x}, \mathbf{y})$ is subsumed by query $Q'(\mathbf{x}', \mathbf{y}')$ if there is a *containment mapping* $\theta : \mathbf{x}' \rightarrow \mathbf{x} \cup N_I$ such that $Q(\mathbf{x}, \mathbf{y})$ is subsumed by $Q'(\mathbf{x}'\theta, \mathbf{y}')$.

We will denote the general query subsumption relation as $\mathcal{K} \models Q \sqsubseteq_\theta Q'$.

2.3.4 A Tableau Algorithm for $\mathcal{SHOIN}$

In this section, we briefly discuss the tableau algorithm for the DL $\mathcal{SHOIN}$. For a detailed description of the algorithm, we refer the reader to [62].

As we have explained above all reasoning problems can be reduced to KB consistency problem. DL tableau-based algorithms decide the consistency of a KB $\mathcal{K}$ by trying to construct (an abstraction of) a model for $\mathcal{K}$. The main elements that characterize a tableau algorithm are [6]:

- An underlying data structure, called the completion graph.
- A set of expansion rules.
- A blocking condition, for ensuring termination.
- A set of clash-triggers, to detect logical contradictions (clashes).

A completion graph $\mathbf{G}$ is a finite directed graph, in which the possible models for the input knowledge base are represented. Each node and edge in $\mathbf{G}$ is labeled with a set of concepts and a set of roles respectively. To decide the consistency of a knowledge base $\mathcal{K}$, the algorithm generates an initial graph $\mathbf{G}$, constructed from $\mathcal{K}$ and repeatedly applies the expansion rules until a clash (i.e. a contradiction) is detected in the label of a node, or until a clash-free graph is found to which no more rules are applicable. The application of a rule may add new concepts to the label of a node, trigger the generation of a new node or cause two different nodes to merge.

A completion graph is called *complete* if it contains a clash, or if no more rules are applicable.

Tableau algorithm for $\mathcal{SHOIN}$ is *non-deterministic* in the sense that there might exist completion rules that yield more than one possible outcome. A tableau algorithm will return consistent iff there exists at least one way to apply the non-deterministic rules such that a complete clash-free graph is obtained.

Termination of tableau algorithms is guaranteed through *blocking*: halting the expansion process when a "cycle" is detected [5]. When the algorithm detects that a path in $\mathbf{G}$ will be expanded forever without finding a clash, then the application of the *generating rules* (the ones that trigger the creation of new nodes in the graph) is blocked in the leaf node x of the path. In such case, we say that x is *blocked* in $\mathbf{G}$. Several bocking strategies have been developed in the literature for different logics (see, for example, [108] [57]).

Reasoning w.r.t. a general TBox $\mathcal{T}$ and a role hierarchy $\mathcal{R}$ can be reduced to reasoning w.r.t. $\mathcal{R}$ alone in the presence of transitive roles and role hierarchies. The entire TBox can be internalized into a single concept description [63] that will be added to every individual in the domain.

For ease of presentation, as usual, we assume all concepts to be in negation normal form (NNF). Each concept can be transformed into an equivalent one in NNF by pushing negation inwards, making use of de Morgans laws and the duality between existential and universal restrictions, and between at-most and at-least number restrictions, [3].

A completion graph for $\mathcal{K}$ is a directed graph $\mathbf{G} = (V, E, \mathcal{L}, \neq)$ where each

node $x \in V$ is labeled with a set of (possibly complex) concepts and each edge $\langle x, y \rangle \in E$ is labeled with a set of role names $\mathcal{L}(\langle x, y \rangle)$ containing (possibly inverse) roles occurring in R. Additionally, we keep track of inequalities between nodes of the graph with a symmetric binary relation $\neq$ between the nodes of $\mathbf{G}$. We will define the exact blocking condition used for $\mathcal{SHOIN}$ below.

In the following, we often use $R \in \mathcal{L}(\langle x, y \rangle)$ as an abbreviation for $\langle x, y \rangle \in E$ and $R \in L(\langle x, y \rangle)$. If $\langle x, y \rangle \in E$, then y is called a *successor* of x and x is called a predecessor of y. *Ancestor* is the transitive closure of predecessor, and *descendant* is the transitive closure of successor. A node y is called an *R-successor* of a node x if, for some R' with $R' \sqsubseteq^* R$, $R' \in \mathcal{L}(\langle x, y \rangle)$; x is called an R-predecessor of y if y is an R-successor of x. A node y is called a *neighbor* (*R-neighbor*) of a node x if y is a successor (R-successor) of x or if x is a successor (R-successor) of y. The set of S-neighbors for a node $x \in \mathbf{G}$ is denoted by $S^{\mathbf{G}}(x)$.

Next, we define and explain some terms and operations used in the expansion rules in more detail:

- **Nominal Nodes and Blockable Nodes** A node x is a *nominal node* if $\mathcal{L}(x)$ contains a nominal. A node that is not a nominal node is a *blockable node.* A nominal $o \in N_I$ is said to be *new in* $\mathbf{G}$ if no node in $\mathbf{G}$ has o in its label.

- **Blocking** A node x is *label blocked* if it has ancestors x', y and y' such that

  1. x is a successor of x' and y is a successor of y',

  2. y, x and all nodes on the path from y to x are blockable,

  3. $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x') = L(y')$, and

4. $\mathcal{L}(\langle x', x) = L(\rangle y', y\langle)$.

In this case, we say that y blocks x. A node is blocked if either it is label blocked or it is blockable and its predecessor is blocked; if the predecessor of a blockable node x is blocked, then we say that x is *indirectly blocked*.

- **Safe Neighbor** An R-neighbor y of a node x is safe if (i) x is blockable or if (ii) x is a nominal node and y is not blocked.

- **Merging and Pruning** Merging a node y into a node x (written Merge(y, x)) in $\mathbf{G}$ yields a graph that is obtained from $\mathbf{G}$ by adding adding $\mathcal{L}(y)$ to $\mathcal{L}(x)$, "moving" all the edges leading to y so that they lead to x and "moving" all the edges leading from y to nominal nodes so that they lead from x to the same nominal nodes; we then remove (or prune) y (and blockable sub-trees below y) from the completion graph. Details of the Merge and Prune operations are provided in [62].

- **Clash** A completion graph $\mathbf{G}$ is said to contain a clash if:

  - There exists a node x in $\mathbf{G}$ s.t. $\{A, \neg A\} \subseteq \mathcal{L}(x)$ for a concept name $A \in N_C$, or,

  - There exists a node x in $\mathbf{G}$ s.t. $(\leq nS) \in \mathcal{L}(x)$ and there are $n+1$ S-neighbors $y_0, \ldots, y_n$ of x with $y_i \neq y_j$ for all $1 \leq i < j \leq n$ for a role $S \in N_R$, or,

  - There are two nodes $x \neq y$ with $o \in \mathcal{L}(x) \cap \mathcal{L}(y)$ for some $o \in N_I$.

Rule	Condition and action
$\sqcap$-**rule**:	If 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, x is not indirectly blocked, and 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$, then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-**rule**:	If 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, x is not indirectly blocked, and 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$, then set $\mathcal{L}(x) = \mathcal{L}(x) \cup C$ for some $C \in \{C_1, C_2\}$
$\exists$-**rule**:	If 1. $\exists S.C \in \mathcal{L}(x)$, x is not blocked, and 2. x has no safe S-neighbor y with $C \in \mathcal{L}(y)$, then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{S\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-**rule**:	If 1. $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2. there is an S-neighbor y of x with $C \notin \mathcal{L}(y)$, then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\forall_+$-**rule**:	If 1. $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and 2. there is some R, with $Trans(R)$ and $R \sqsubseteq^* S$, and 3. there is an R-neighbor y of x with $\forall R.C \notin \mathcal{L}(y)$, then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{\forall R.C\}$
$\geq$-**rule**:	If 1. $\geq nR \in \mathcal{L}(x)$, x is not blocked, and 2. there are not n safe S-neighbors $y_1, \ldots, y_n$ of x with $y_i \neq y_j$ for $1 \leq i < j \leq n$, then create n new nodes $y_1, \ldots, y_n$ with $\mathcal{L}(\langle x, y_j\rangle) = \{R\}$, $\mathcal{L}(y_j) = \{C\}$, and $y_j \neq y_k$ for $1 \leq j < k \leq n$
$\leq$-**rule**:	If 1. $\leq nR \in \mathcal{L}(x)$, x is not directly blocked, and 2. x has more than n S-neighbors and there are two S-neighbors y and z s.t. $y \neq z$ does not hold then 1. If y is a nominal node, then Merge(z, y), 2. else if z is a nominal node or an ancestor of x, then Merge(y, z), 3. else Merge(z, y)
$\mathcal{O}$-**rule**:	If 1. For some $o \in N_I$, there are 2 nodes x, y with $o \in \mathcal{L}(x) \cap \mathcal{L}(y)$ and not $x \neq y$ then Merge(x, y)
$\mathcal{O}?$-**rule**:	If 1. $\leq nR \in \mathcal{L}(x)$, x is a nominal node, x is an S-successor of a blockable node y and not $x \neq y$ 2. there is no m such that $1 \leq m \leq n$, $\leq mS \in \mathcal{L}(x)$, and there exist m nominal S-neighbors $z_1, \ldots, z_m$ of x with $z_i \neq z_j$ for all $1 \leq i < j \leq m$. then 1. guess m with $1 \leq m \leq n$ and set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{\leq mS\}$ 2. create m new nodes $y_1, \ldots, y_m$ with $\mathcal{L}(\langle x, y_i\rangle) = \{S\}$, $\mathcal{L}(y_i) = \{o_i\}$ for each $o_i \in N_I$ new in $\mathbf{G}$, and $y_i \neq y_j$ for all $1 \leq i < j \leq m$

Table 2.2: Tableau expansion rules for $\mathcal{SHOIN}$

The completion graph for a $\mathcal{SHOIN}$ KB $\mathcal{K} = (\mathcal{A}, \mathcal{T}, \mathcal{R})$ is initialized as an arbitrarily connected graph of nominal nodes, each representing a nominal (individual) used in $\mathcal{A}$ or $\mathcal{T}$. If a is an individual used in $\mathcal{K}$ then the corresponding nominal node in $\mathbf{G}$ will be named r_a and the label of this node will be initialized as $\mathcal{L}(a) = \{a\} \cup \{C \mid C(a) \in \mathcal{A}\}$. The edges between two nodes a and b will be initialized as $\mathcal{L}(\langle a, b\rangle) = \{R \mid R(a, b) \in \mathcal{A}\}$.

Then, the expansion rules shown in Table 2.2 are applied in succession to build the graph, each adding new nodes or edges (and/or labels resp.). Note that some of the expansion rules are non-deterministic. For example, if the disjunction $C \sqcup D$ is present in the label of a node, the algorithm chooses either C or D to be added to the node label. Therefore, in practice, each possible expansion should be tried in turn until a fully expanded and clash-free graph is found, or all possibilities have been shown to lead to contradictions. Searching non-deterministic expansions is the main cause of intractability in the tableaux algorithm.

2.4 Semantic Web Services

OWL-S [96] provides a set of OWL ontologies to describe Web Services in a more expressive way than allowed by WSDL. The features of the Web Service, e.g. message types, constraints and capabilities, are defined using the terms from Web Ontologies. OWL-S partitions the semantic description of a web service into three components: the service profile, process model, and grounding. The *Service-Profile* describes what the service does by specifying the input and output types,

preconditions and effects. The *ProcessModel* describes how the service works; each service is either an *AtomicProcess* that is executed directly or a *CompositeProcess* that is a combination of subprocesses (i.e., a composition). The *Grounding* contains the details of how an agent can access a service by specifying a communications protocol, parameters to be used in the protocol, and the serialization techniques to be employed for the communication. The similarities between OWL-S and other technologies may be briefly expressed as follows: the ServiceProfile is analogous to yellow-page- like advertisements in UDDI, the ProcessModel is similar to the business process model in BPEL4WS, and the Grounding is a mapping from OWL-S to WSDL. The main contribution of OWL-S is the ability to support richer descriptions of the services and the real world entities they affect in such a way as to support greater automation of the discovery and composition of services. In Chapter 4, we will provide a more detailed analysis of OWL-S and discuss how it relates to HTN-DL.

2.5 AI Planning

Most of the planning approaches rely on a general model, the model of state-transition systems. In a state-transition system there are finite or recursively enumerable set of states, actions and events along with a transition function that maps a state, action, event tuple to a set of states. Given a state transition system, the purpose of planning is to find which actions to apply to which states in order to achieve some objective, starting from some given situation.

Classical planning is mainly based on the initial modeling of the STRIPS [29] system. In this representation a state is represented by a set of ground literals expressed in a first-order language. An action is an expression specifying which first-order literals belong to the state in order for the action to be applicable, and which literals the action will add or remove in order to make a new world state. An atom p holds in state s iff $p \in s$. If g is a set of literals with variables, s satisfies g (denoted $s \models g$) when there is a substitution σ such that every positive literal of $\sigma(g)$ is in s and no negated literal of $\sigma(g)$ is in s.

In classical planning, a planning operator is a triple $o = (N, P, E)$, where N, name of the operator, is a syntactic expression $n(x_1, \ldots, x_n)$ and n is a unique operator symbol and $x_1, \ldots, x_n$ is the inputs of the operator, P is the precondition of the operator expressed as a conjunction of (possibly unground) set of literals. E is the effects of the operators which can be positive or negative, i.e. E^+ (generally referred as the add list) represents the set of literals that will be added to the state and E^- (generally referred as the delete list) represents the set of literals that will be removed from the state. An operator o is applicable in a state s when the preconditions are satisfied in the state. Most planners represent the world state with a relational database and thus precondition evaluation is straight-forward. Applying the effects of an operator is done by adding or deleting entries from the database.

This representation is insufficiently expressive for some real domains. As a result, many language variants have been developed. Action Description Language (ADL) [100] is an important variation. ADL extends STRIPS representation by explicitly including negative literals in the state, having conditional effects for oper-

ators and allowing existential variables and disjunctions in goal formulas. Penberthy and Weld [101] developed a partial order planning algorithm named UCPOP [102] to handle a significant subset of ADL action representation.

HTN planning is similar to classical planning in that each world state is represented by a set of literals and each action corresponds to a state transition. However, HTN planners differ from classical AI planners in what they plan for, and how they plan for it. The objective of an HTN planner is to produce a sequence of actions that perform some activity or task. The description of a planning domain includes a set of operators similar to those of classical planning, and also a set of methods, each of which is a prescription for how to decompose a task into subtasks. Planning proceeds by using methods to decompose tasks recursively into smaller and smaller subtasks, until the planner reaches primitive tasks that can be performed directly using the planning operators.

Here we will provide a formal description of a simplified version of HTN planning formalism as presented in more detail in [36]. This version is called Simple Task Network (STN) planning in [36] and is slightly less general than the formalization of HTN planning presented in [26]. However, even with this simplification, HTN planning is more expressive than classical planning [36] and allows very efficient implementations, as in SHOP2 system [93].

Formally, a task is an expression of the form $t(x_1, \ldots, x_n)$ where t is the *task symbol* and $x_1, \ldots, x_n$ are parameters of the task given as an ordered list. Task symbols are categorized into two disjoint sets: *primitive* and *nonprimitive*. By definition, every operator symbol is a task symbol.

An HTN method is a tuple $m = (N, T, P, w)$. N, the name of the method, and P, the precondition of the method, is defined similar to operators. T is a nonprimitive task denoting the task this method accomplishes. w is a *task network* which contains tasks, along with ordering constraints on these tasks. The ordering between the tasks can be partial or total. The elements of the task network are called subtasks of m.

Task matching is deciding which actions can be used to possibly accomplish a task. An operator $o = (N, P, E)$ *matches* a task t if $N = t$. A method $o = (N, T, P, w)$ *matches* a task t if $T = t$. An operator o (resp. method m) is applicable to a primitive (resp. nonprimitive) task t in state S if o (resp. m) matches t and P^o (resp. P^m) is satisfied in S.

An HTN planning problem is the triple (s, w, D) where s is the initial state, w is the initial task network, and D is the planning domain that consists of operator and method descriptions. A forward-search HTN planner starts with the initial task network and selects a task that has no predecessors. If the task is primitive, an applicable operator is found and added to the plan. For nonprimitive tasks, an applicable method is found and the selected task is replaced with the subtasks of the method. Planning continues recursively until there are no tasks left in the network.

Chapter 3

Coupling Planning with Description Logics: HTN-DL

In this chapter we first discuss how AI planning techniques can be used for Web Service composition problems. We examine the common issues and problems that arise when we try to model Web Services in a planning domain. Then we present HTN-DL a formalism that combines HTN planning with Description Logics to overcome these problems.

3.1 AI Planning and Web Service Composition

Several Semantic Web Services languages describe services in ways amenable to planning since services have preconditions and effects that are expressed as logical conditions. Using this similarity, it is possible to treat Web Services as planning operators and use a causal planner in the style of STRIPS to generate Web Service compositions. With this approach each Web Service is first translated to a planning operator, the objective is expressed as a logical condition, and the planner generates a plan which is essentially a sequence of Web Service instances — that is, a sequential composition that causes the goal condition to be true upon execution.

Unfortunately, this direct encoding of Web Service composition as a planning problem itself has several problems. First, the typical logic for expressing preconditions and effects in a planning system has a radically different expressiveness than

the Web ontology languages do. In a planning system, the state of the world representation contains only facts about the world but not axioms.[1] However, such axioms are essential in Web Service composition problems to help the system figure out relations between complex concepts, such that it is possible to relate terms from different ontologies.

Another issue is the Closed World Assumption (CWA) that AI planning systems rely on. If a fact is not contained in the local knowledge base, then it is simply assumed to be false. However, this assumption is not realistic in the Web context as we cannot expect to have all the relevant information in our local knowledge base. When making decisions we need to consider the known facts and possibilities that are consistent with these facts.

The planner having incomplete information needs to gather some information in order to solve the composition problem. It is necessary that information-gathering and composition generation is interleaved so that a decision can be made to include which service in the composition. Given the Web's size and nature, it is likely that trying to gather all the possible information will be wasteful at best and practically impossible in the common case. The relevance of possible information should be determined by the context of the composition problem and possible combinations the planner is considering, which means that it makes sense to gather the information at that point.

This also requires the planning system to distinguish information-providing

[1] Sometimes axioms in the form of Horn rules are also allowed, but this is still far from the expressivity of OWL, for example.

services from world-altering services. We need to describe if executing a service will provide just information or have some other effects on the world. For example, learning the available airline schedules provides us some information we were not aware of. On the other hand, buying the plane ticket has some effects on the world, e.g. one less available seat in that flight, less money in the banking account, etc.

The planner can execute the information-providing services freely as it does not change anything in the world other than our knowledge.[2] However, executing a world-altering service requires much more caution because it might commit the user to do something that is not desired. Buying an airplane ticket without making sure that there is an available hotel at the specified time would yield very undesirable results.

Note that a world-altering service might also provide some information as a result of execution. Actually, any service that has an output is providing some information. For the flight reservation service, the effect of the service is booking the ticket and the output is a confirmation code associated with the reservation. Therefore, the same service can have both world-altering effects and knowledge effects that need to be identified.

Representing a Web Service as a planning operator has two other problems: 1) It is not possible to describe the internal structure of composite services. 2)

[2] Actually, this is not exactly correct because an information gathering process might affect other things such as other parties' knowledge. For example, supplying your address and social security number to a Web Service has the effect that the other party has knowledge about your personal identity. One would want to be careful before executing such information-providing services. However, such issues are beyond the scope of this thesis and will not be addressed here. We will simply assume that the information sent to an information-providing service is only used to return an answer and is not used or stored for another purpose.

A planning operator has only information about preconditions and effects, but we need to express more information about services, e.g. the provider of the service, credentials of the provider, user ratings about the service, etc. Let us examine these two cases in more detail.

Some Web Services have a composite structure, i.e. they are constructed from other (possibly composite) services. For example, such a composite service might describe several different steps of interacting with a Web site. Alternatively, composite services can act as reusable and customizable template descriptions outlining the standard operating procedures for accomplishing a task. It is not possible to treat these composite services as atomic because there might be (possibly non-deterministic) choices inside the composite service that will change the way service affects the state. The planner needs to consider the internal structure of the services and plan accordingly.

The non-functional attributes of a service play a very important role when there are many services with similar functionality and capabilities. The only distinguishing features between services might be attributes such as who is providing the service, what kind of credentials they have, what kind of security policies are being used in the communication. Unfortunately, a planning operator cannot represent such information.

3.2 HTN-DL

3.2.1 Overview

HTN-DL is a planning formalism that combines HTN planning formalism with Description Logic (DL) representation to overcome the problems described in the previous section. The DL is used to describe both actions and states with an expressive knowledge representation language.

HTN planning, as it stands, looks promising to tackle the Web Service composition problems. The hierarchical structure of HTN planning domains (see Section 2.5) can conveniently describe composite service descriptions. Composite Web Services can be mapped to HTN methods whereas atomic Web Services are mapped to HTN operators (we will discuss the details of this mapping in Chapter 4). HTN-style domains fit in well with the loosely coupled nature of Web Services: different decompositions of a task are independent so the designer of a method does not have to have close knowledge of how the further decompositions will go or how prior decompositions occurred.

Nevertheless, HTN planning suffers from most of the problems discussed in the previous section, e.g. state representation is simple, Closed World Assumption is adopted, there is no notion of outputs or knowledge effects, no attributes of actions other than preconditions and effects are described.

Furthermore, there are some other restrictions and limitations specific to HTN planning that make it difficult to apply to Web Service composition problems:

- In HTN planning, a task is identified only by its name and the number of arguments it has. This is all the information to identify what kind of functionality this task accomplishes. Naturally, it is not possible to express or infer relations between different tasks.

- There is a strict separation between *primitive* and *nonprimitive* tasks which is not applicable to Web Services. Due to this separation, we cannot have both a method and an operator achieving the same task. However, this is a very common case for Web Services as we can have both an atomic and a composite service with the exact same functionality. A simple example is online shopping services. Some services request you in one step to send the purchase order with all necessary shipping and billing information; some other services ask you to create an account (or login to an existing account), add the items to a shopping cart, and proceed to checkout. In the end, the task "buying an item" is essentially the same in both cases.

- In HTNs there is a one-to-one mapping between operators and primitive tasks. Hence, there can only be one operator that accomplishes any given primitive task. Clearly, this assumption is also too restricted for Web Services.

In order to overcome these problems, HTN-DL combines HTN planning with DL ontologies. The key differences of HTN-DL compared to classical HTN planning systems can be summarized in the following categories:

- *Task descriptions* Tasks are described using ontologies and matching tasks with operators and methods is done based on the task ontology. More specifi-

cally, task symbols are represented as DL concepts and operators, and methods are represented as instances. Task matching is partly reduced to the instance retrieval problem in DLs. In addition, preconditions and effects are associated with tasks, so that more information about the task is provided which is also used to determine matching services.

- *Operator/method descriptions* Operator and method definitions use DL queries to describe the conditions in preconditions and effects. Outputs of services, which are not traditionally considered in planning systems, are represented as existential variables in effect descriptions. Knowledge effects of an action are expressed separately so that information-providing services can be distinguished from world-altering services.

- *State representation* In HTN-DLthe state of the world is described as a DL knowledge base. This allows one to use a very expressive knowledge representation language represent the known information about the world. As traditional in DLs, the Open World Assumption is adopted in reasoning.

Before describing the formal syntax and semantics of the HTN-DL let us give a brief overview of the formalism with a simplified Web Service composition example. Suppose a conference Web site is providing a Web Service that will help the attendants make their travel arrangements to the conference. Conference organizers create composite Web Service descriptions with several components to handle different tasks: registration to the conference, as well as making arrangements for both accommodation and transportation. The workflow of such a composite service

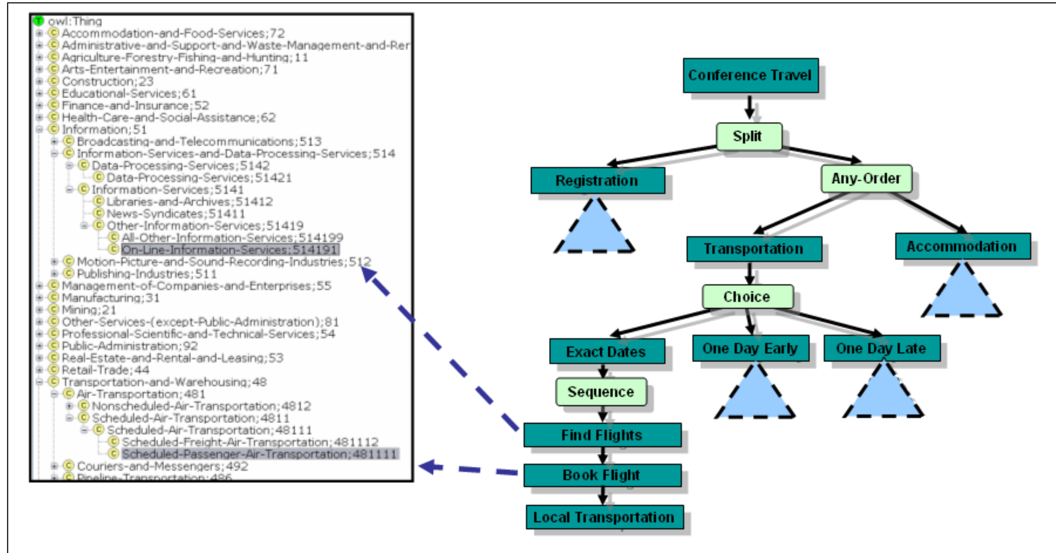

Figure 3.1: A composite Web Service provided by a conference Web site that makes the arrangements for people coming to the conference. Some parts of the service, e.g. making travel arrangements, needs to be done by third party services. These parts are described in abstract using an ontology so the user agent reading this description can find and plug-in an appropriate service for these steps

is shown in Figure 3.2.1 as a flow chart.

The registration service is directly provided by the conference organizers, but the services to make accommodation and travel arrangements are provided by external parties. The conference organizers do not want to commit to any specific travel agent service at these steps so that each user can customize this template according to their needs and preferences. At design time, such components can be described as abstract services; and, at run time, they will be matched with concrete services by the planner.

In HTN-DL, such abstract services are represented as HTN-DL tasks. An HTN-DL task is primarily described as a concept in a task ontology. This concept represents a *service category*, which means that services belonging to this category can provide the desired functionality. For example, in the example shown

in Figure 3.2.1, the abstract service *Book Flight* is associated with the category *Scheduled-Passenger-Air-Transportation* from the North American Industry Classification System (NAICS) ontology. Any service which is inferred to belong to this category would be considered as a possible candidate. Of course, this category need not be a named concept; it can be a complex concept that describes additional restrictions on the service. These restrictions would be typically defined on the non-functional attributes of the service, e.g. its rating, provider, QoS attributes, etc.

In addition to the service category, tasks can have preconditions and effects describing when a service is executable and what will happen after execution. However, when matching a task with a concrete service, the preconditions and effects do not need to be an exact match. In general, the matching service can have a more generic precondition and a more specialized effect. Since preconditions and effects are described as DL conjunctive queries, matching can be done by query subsumption (See Section 2.3.3 for the definition of query subsumption). Condition C_1 is more generic (resp. specialized) than condition C_2, if the condition C_1 is subsumed by (resp. subsumes) the condition C_2.

In Table 3.1, we show an example of precondition and effect expressions for the *Book Flight* task and a service from an imaginary travel agency SemanticTravel. The precondition of the task says that we want to only book flights originating from the US. The precondition of the SemanticTravel service indicates that it sells plane tickets originating from either US or Canada. This is considered a match because the capability of the service covers (and does more than) the functionality

we specified in the task. The effect of the *Book Flight* task requires that, at the end of the Web Service execution, the customer should own a ticket. The description of SemanticTravel service says that the customer will own a non-refundable airplane ticket. This is a more specialized effect expression which means accomplishing this effect will also accomplish the effect we desire. Therefore, SemanticTravel service is a valid match for the *Book Flight* task.

After the matching is done and some candidate services are found we still need to check their preconditions to verify that necessary conditions to use these services is met. In HTN-DL state information is also represented using DL ontologies so precondition evaluation is done by DL query answering. Note that, classical planning systems typically adopt Closed World reasoning for precondition evaluation. However, due to standard DL semantics, we adopt Open World reasoning in HTN-DL.

Some of the matching services may not have any effects on the world but just provide information to the users. A service that returns plane schedules is one such service. Normally a planner does not execute any of the actions put into the partial plan because it is not guaranteed that a final plan can be generated. For example, booking a hotel before ensuring that a plane ticket can be found would yield undesirable outcomes. But for a service that have only knowledge effects this

	Book Flight Task	SemanticTravel Service
Precondition	$USAirport(departAirport)$	$USorCanadaAirport(departAirport)$
Effect	$owns(cust, t) \wedge Ticket(t)$	$owns(cust, t) \wedge PlaneTicket(t)$ $\wedge\ NonRefundable(t)$

Table 3.1: Preconditions and effects of a matching task and a concrete service

is not an issue since it can be repeatedly executed without changing the state and committing the user to any other action. In HTN-DL, purely information providing services are executed during planning to gather information that can be used in subsequent steps.

In the next sections, we will provide the formal syntax and semantics of HTN-DL. Note that, there are many different DLs with varying expressivity investigated in the literature. In what follows, we define the general characteristics of HTN-DL in a such a way that a different DL can be *plugged-in*. There is no requirement to a DL with specific expressivity. Furthermore, it is also possible to use DLs with different expressivity in state representation and task ontology. In Section 3.3, we will discuss how the choice of DL expressivity affects some of the algorithms.

3.2.2 Syntax

We start by defining what a state is.

Definition 3.1 (State) *State $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ is a DL ontology, where $S_{\mathcal{A}}$ is an ABox (set of ground DL-literals), $S_{\mathcal{T}}$ is a TBox (set of concept inclusion axioms), and $S_{\mathcal{R}}$ is an RBox (set of role inclusion axioms and transitivity axioms).*

We consider that the planner's local state is an incomplete description of the world. There are possibly many interpretations that satisfy all the assertions and axioms in the state. One of these interpretations is the exact description of the state of the world, denoted by W. However, without having complete information, the planner does not know which interpretation is the correct one.

The component $S_{\mathcal{A}}$ is similar to what is traditionally considered to be a *state* in planning. It is a collection of facts known about the current state of the world. Since we do not have complete information, we cannot use Closed World Assumption (CWA) and we store both positive and negative facts. We employ Open World Assumption (OWA) when querying the state in accord with with the standard DL semantics.

The combination of $S_{\mathcal{T}}$ and $S_{\mathcal{R}}$ constitute what is traditionally called *state constraints*, sometimes also referred as *domain constraints* or *domain axioms*. These axioms reflect the knowledge about the world that holds for every possible world. A state would be considered *inconsistent* if there are assertions in the state contradicting these axioms. In this manner, state constraints help us to detect the integrity of the agent's local knowledge. In addition, these axioms may be used to infer new facts from the asserted facts in the state.

We assume that actions we perform can change the state only by modifying the facts in $S_{\mathcal{A}}$ but actions are not allowed to change the state constraints. For example, if the state constraints contain the inclusion axiom $Course \sqsubseteq \forall takes^{-}.Student$ saying that only students can take courses, execution of no service can change this axiom. But the fact that Bob takes the course CS-101, denoted with the ABox assertion $takes(\texttt{Bob}, \texttt{CS-101})$, can be changed with the execution of a service, e.g. by dropping the course using the service provided by the registrar's office.

In the real world, state constraints might change over time. For example, at some point the regulations can be modified such that people who are not students will be allowed to take courses. We assume that such changes will be done outside

the HTN-DL formalism; that is, the changes to the state constraints will be done externally and then the new state information can be fed to the planner to generate plans.

In a DL that allows nominals, the separation between the state facts and state constrains is not well-defined as we can encode all the ABox assertions of $S_{\mathcal{A}}$ as TBox axioms in $T_{\mathcal{A}}$, e.g. ABox assertion $C(a)$ is equivalent to the inclusion axiom $\{a\} \sqsubseteq C$. Although these two statements are semantically equivalent, since we require that state constraints hold in every state, encoding an ABox assertion as a TBox axiom will have a different impact. This encoding models what is called a *rigid relation* [37] in the planning literature; that is, relations that cannot be changed by any action. For example, the relation $hasBorder(\texttt{USA}, \texttt{Canada})$ is a rigid relation that cannot change over time. In a classical planning system, it is to ensured that rigid relations will always persist by disallowing those relations in the effects of actions. When state constraints are involved, ensuring this is not straight-forward as the ramifications of an action might influence relations that are not explicitly stated in the effect expression. By encoding rigid relations as state constraints, we can guarantee that they will not be affected by the actions.

As in the original HTN formalism, we have two types of actions: *operators* that are atomic actions with no internal structure and *methods* that are composite actions which can further be decomposed into smaller parts.

Definition 3.2 (Operator) *An operator is described as* $o = (N, I, O, P, E_W, E_K)$*. N is the operator name, I is the set of input parameters, O is the set of output*

parameters. P is the preconditions of the action in the form of a boolean DL conjunctive query. E_W and E_K are the effects of the action, again in the form of a boolean DL conjunctive query, and they describe how the state changes after execution. More specifically, E_W is the world-altering effects, i.e. it describes how the world state W will change after execution of o. E_K, on the other hand, is the knowledge effects, i.e. it describes how planner's local state S will change.

HTN-DL operator definition differs from the definition of standard HTN operators in several ways. First, the ordering of inputs for an HTN-DL operator is not significant whereas in HTN planning the order of the inputs determine their function. Second, we have the notion of outputs in HTN-DL which can be thought as existential variables referred in the effects description. The value of an output variable can only be learned after executing the operator. Effects of an operator is similar to HTN case; it describes the changes that will happen in the world. Knowledge effects, on the other hand, do not describe changes in the state of the world but explains how the mental state of the agent will change after the execution of this action. By executing the action, the planner will learn the values corresponding to the output variables of the action such that the conditions in the knowledge effects hold. In other words, after executing an action that has only knowledge effects, the world stays in the same state W but planner's state S is expanded to better approximate W.

Although knowledge-producing effects are explicitly specified, HTN-DL does not have the notion of *knowledge preconditions* in operator definitions. This is

simply because we assumed that the preconditions for actions do not depend on the knowledge of an agent. The preconditions of an action may only depend on the current state. This is similar to the *knowledge-free Markov* assumption in [40]. This assumption is true for Web Services because when we are executing a remote Web Service the Web Service provider does not have any information about our knowledge other than the information we supply via the inputs of the service. In this sense, inputs of an action encode knowledge preconditions because the agent needs to *know* the input values in order to execute the action.

The precondition expression P is in the form of a boolean DL conjunctive query. This expression may contain variables from I which will be substituted with the given input values before evaluation. The additional variables in P are simply treated as existential variables.

We do not associate types with the input or output variables. As usual, type of a parameter is simply a unary predicate, i.e. a DL concept, and can be encoded as an additional conjunct in the precondition expression. Similarly, the output types can be encoded as knowledge effects.

Example 3.1 *Consider a course registration service. This service, given the student ID, the class and the term to register for, registers the student to the course. This service can only be used if the student has completed the prerequisites of the class and there are open seats. The following operator definition describes such a service. Note that, the operator has two outputs, a transaction ID that can be used as a reference number for the student's proof of registration and a timetable for the*

student. Upon execution, we will learn information about these outputs and what we will learn is described as the knowledge effects of this operator.

$$
\begin{aligned}
o =& (registerClass, \{studentID, class, term\}, \{transactionID, timeTable\},\\
& completedPrereq(studentID, class) \wedge open(class, term),\\
& takesCourse(studentId, class),\\
& Transaction(transactionID) \wedge TimeTable(timeTable))
\end{aligned}
$$

Definition 3.3 (Task Description) *A task has the form $t = (N, I, O, P, E_W, E_K)$ where N is the task symbol, I is the set of input parameters, O is the set of output parameters, P is the precondition expression, E_W is the effect expressions, and E_K is the knowledge effects similar to operator descriptions.*

A task is an abstract description of a functionality. Intuitively, the task symbol represents the functionality required and inputs/outputs are used to specify the data flow between tasks. Precondition and effects are used to provide additional information about how the inputs and outputs of a service are related.

Note that, the definition of an HTN-DL task is significantly different then an HTN task which is described solely by its name and its parameters. The additional information we provide in the task descriptions will allow us to define a much more flexible task matching definition as we will explain in the next section in more detail.

Example 3.2 *Example 3.1, describes a concrete action we can use for registering a course. The following example shows a generic task studentRegistration describing*

a similar functionality. Task definition is slightly different: It has one less output and uses a different vocabulary to describe preconditions and effects. We will later explain how these two descriptions can be matched.

$$\begin{aligned} t = (&studentRegistration, \{studentID, course, semester\}, \{transactionID\}, \\ &completedPreReq(studentID, class) \wedge haveSeat(course, semester), \\ &takesCourse(studentId, course), \\ &Tranaction(transactionID)) \end{aligned}$$

Before we describe methods, first we define *task networks* which will be used to construct methods.

Definition 3.4 (Task Network) *A task network is an acyclic directed graph in the form $w = (\eta, \epsilon, \lambda)$ where η is the set of nodes, ϵ is the set of edges, and λ is is a list of parameter binding constraints. Each node $u \in \eta$ contains a task t_u. Parameter bindings are in the form $a \leftarrow b$ such that*

- *a is either a variable symbol or a a tuple $\langle u, p \rangle$ where u is a node in η and p is an input parameter associated with task t_u.*

- *b is either a variable symbol, a constant or a tuple $\langle v, q \rangle$ where v is a node in η and q is an output parameter associated with task t_v.*

The edges of w define a partial ordering of η. That is, $u \prec v$ if the there is a path from u to v. These ordering constraints specify which task will be achieved before which task. If the ordering is total we say w is *totally ordered*.

The parameter bindings in our task network definition is something novel to HTN-DL. The purpose is simply to define the data flow between different tasks in the network and possibly the parameters of the enclosing method. In classical HTN, the data flow is established by using the same variable symbols in different task occurrences and then let the unification algorithm take care of the rest. This method is not applicable in HTN-DL because the order of parameters in task descriptions have no meaning (recall that input and output parameters are defined as sets not ordered lists). For this reason, we use parameter binding constraints that explicitly specify the occurrence of a task (via the task node) and the name of the parameter.

Note that the output of one task can be specified as input to another task in the parameter binding constraints. This binding implies an ordering constraint between these two task occurrences as we need to achieve the task with the output before the other task. Without loss of generality, we will assume that such implicit ordering constraints are already deducible from ϵ. Note that, we still require the task network to be acyclic so any cyclic parameter binding constraint is not allowed.

Example 3.3 *Course registration has additional steps such as charging the student account and sending a notification message. The following task network describes these steps involved in the process:*

$$
\begin{aligned}
w =& (\{u_1 = studentRegistration, u_2 = updateStudentsBill, u_3 = notify\},\\
& \{(u_1, u_2), (u_2, u_3)\},\\
& \{\langle u_1, studentID\rangle \leftarrow SID, \langle u_1, course\rangle \leftarrow CourseName, \langle u_1, term\rangle \leftarrow term,\\
& \langle u_2, transactionID\rangle \leftarrow \langle u_1, transactionID\rangle, \langle u_3, bill\rangle \leftarrow \langle u_2, bill\rangle\\
& \langle u_3, studentID\rangle \leftarrow SID, receipt \leftarrow \langle u_1, transactionID\rangle\})
\end{aligned}
$$

This is a totally ordered task network with three task nodes u_1, u_2 and u_3 that correspond to the tasks `studentRegistration`*,* `updateStudentBill` *and* `notify`*, respectively. According to the parameter bindings the input for the task node u_2 is the output of the node u_1 which is the transaction id returned by* `studentRegistration`*. Similarly the inputs for u_3 come from the outputs of u_1 and u_2. The inputs of u_1 come from the variables SID, $CourseName$ and term. Furthermore, the output of u_1 is bound to the variable receipt.*

Definition 3.5 (Conditional Task Network) *A conditional task network is a construct of the form $[(C_1 : \tau_1) \ldots (C_n : \tau_n)]$ where each C_i is in the form of a boolean DL query and each τ_i is a task network.*

Intuitively, a conditional task network is a nested *if-then-else* structure describing different ways to decompose a method under different conditions. The conditions C_i are in the form of a boolean query. We find the smallest i such that C_i is true in the current state. We can also use a condition such as $\top(a)$ where a

is an arbitrary individual to express a condition that will always be satisfied. This might be useful for the cases if we want the last branch to be unconditional, e.g. representing the *else* part of an *if-then-else* structure.

Definition 3.6 (Method) *A method has the form* $m = (N, I, O, P, V, E_W, E_K, \Gamma)$ *where* N*,* I*,* O*,* P*,* E_W*, and* E_K *are defined similar to the operators. The additional element* V *is a subset of local variables mentioned in the expression* P*.* Γ *is a conditional task network such that for every parameter binding* $a \leftarrow b$ *in* Γ *if* a *is a variable then* $a \in O$ *and if* b *is a variable then* $b \in I \cup V$*. Furthermore, for each output* $o \in O$ *there is at most one binding* $o \leftarrow b$ *in the parameter bindings of every task network in* Γ*.*

A method description is syntactically very similar to an operator description but in addition it contains a conditional task network component. A method achieves a task by accomplishing the subtasks defined in that task network whose condition is satisfied.

The local variables V of a method are simply some variables that will be bound by the precondition expression. The values are then used as input parameters to the subtasks in the network Γ. Of course, in some cases, there might be multiple different bindings that satisfy the precondition expression. The planner would consider all such bindings while searching for a plan. Note that, the conditions defined inside the task network Γ cannot be used to assign any values to the variables in V since they are only boolean queries.

Methods might also have effects associated with them but they should not be confused by the *high-level effects* as used in some HTN systems [124]. Following the HTN formalism presented in [27], we treat such effects as *postconditions* that need to hold after the last task in Γ is achieved. State can only be changed by operators. The effects of a method just defines some conditions that will hold after all the tasks in Γ are achieved regardless of in which order or with which actions those tasks are achieved.

Example 3.4 *The registration process is slightly different for students who work as a graduate assistant as their payment is waived. The following is a method that combines the task network w of Example 3.3 with this additional condition.*

$$
\begin{aligned}
m = (®ister, \{SID, CourseName\}, \{term\}, \{receipt\},\\
&nextTerm(term) \wedge open(courseName, term),\\
&takesCourse(SID, course), \top,\\
&[(GraduateAssistant(SID) : (\{u_1 = studentRegistration\}, \emptyset, \lambda));\\
&(\top : w)])
\end{aligned}
$$

where w is the task network in Example 3.3 and λ is defined as

$$
\begin{aligned}
\lambda = \{&\langle u_1, studentID\rangle \leftarrow SID,\ \langle u_1, course\rangle \leftarrow CourseName,\\
&\langle u_1, term\rangle \leftarrow term,\ receipt \leftarrow \langle u_1, transactionID\rangle\}
\end{aligned}
$$

Definition 3.7 (Task Ontology) *A* task ontology T_{ont} *is a DL ontology where*

task symbols are used as concept names and operator and method names are used as individual names.

There are several functions of the task ontology. First, it describes how tasks are related to each other via the subsumption hierarchy. For example, `buy-new-book` and `buy-used-book` tasks can be described as specializations of the more general task `buy-book`. This would simply be achieved with a concept inclusion axiom in the task ontology.

The task ontology also describes which operators or methods can be used to achieve a certain task. For example, a membership inclusion axiom can state that `acme-book-service` is an instance of the `buy-new-book` concept. There can be additional properties described about services in the task ontology. For example, the task ontology might contain the following assertions about `acme-book-service`:

$$providedBy(\textit{acme-book-service}, \textit{acme-corporation}).$$

$$locatedIn(\textit{acme-corporation}, USA).$$

$$CertifiedSeller(\textit{acme-corporation}).$$

Suppose there is also a task describing any functionality provided by a certified organization located in USA:

$$CertifiedUSSeller \equiv \exists providedBy.CertifiedSeller \sqcap locatedIn.\{USA\}$$

Looking at the task ontology one can conclude that `acme-book-service` is an instance of $CertifiedUSSeller$ and can be used to achieve this task.

A planning domain is simply a combination of operator and method descriptions coupled with a task ontology.

Definition 3.8 (Planning Domain) *A* planning domain *D is defined as the triple (O, M, T_{ont}) where O is the set of operator descriptions, M is the set of method descriptions, and T_{ont} is the task ontology.*

Finally, we give a formal definition of the planning problem.

Definition 3.9 (Planning Problem) *A* planning problem *P is a triple (S, w, D) where S is the initial state, w is the task network to plan for, and D is the planning domain.*

3.2.3 Semantics

In this section, we provide an operational semantics for HTN-DL. We start by defining the semantics for states. Let us first define the *consistency* of a state by reiterating the consistency definition of a DL knowledge base.

Definition 3.10 ((In)consistent State) *A state $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ is consistent if there is an interpretation $\mathcal{I}$ that satisfies all the assertions and axioms in S. The state is inconsistent if there is no such interpretation.*

An inconsistent logical theory entails everything which would cause a lot of problems in our case. If the planner's state is inconsistent then there would be

no point in planning because every action would be considered possible. For this reason, we will make the following simplifying assumption.

Assumption 3.1 (Global Consistency Assumption (GCA)) *The complete knowledge about the world* $W = \langle A_W, \mathcal{T}_W, \mathcal{R}_W \rangle$ *is always consistent.*

Such an assumption is not very meaningful especially in the context of Web. It is inevitable that any agent interacting with Web will encounter contradicting information. However, dealing with such inconsistency is beyond the scope of this thesis. See Section 9.4 for a discussion on this subject.

It is important to emphasize that GCA not only requires the initial state to be consistent but also ensures that world state will remain consistent after execution of any action. That is, if there is a service whose effects will cause the state to become inconsistent then we will conclude that this service is not executable in real world.

We also assume that planner always has correct knowledge about the world.

Assumption 3.2 (Local Correctness Assumption (LCA)) *Planner's incomplete knowledge about the world, denoted by* S*, is always correct. Formally, we say* $W \models S$*, that is, every entailment of* S *is also an entailment of* W*.*

The preconditions of operators and methods as well as the conditions inside task network are expressed in the form of conjunctive DL queries (see Section 2.3). Due to Open World Assumption a condition might have three different truth values:

Definition 3.11 (Truth Value of Conditions) *Let* $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ *be the current state and* Q *be a condition expressed as boolean conjunctive query. We say that*

condition Q is **`true`** *at state S iff $S \models Q$. Condition Q is* **`false`** *at state S iff $S \models \neg Q$). Truth value of the condition is* **`unknown`** *if Q is neither true nor false at state S.*

Note that, due to incomplete information, the planner might conclude that a condition Q is not satisfied at the current state S although Q is true at W. This means some actions that are actually possible will not be considered by the planner which is a consequence of incomplete information. However, it is important to emphasize that the converse does not cause any problem. That is, whenever Q is true at S, it is also guaranteed to be true at W due to LCA.

Definition 3.12 (Task Matching) *Let $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ be the current state, $D = (O, M, T_{ont})$ be the planning domain, and $t = (N^t, I^t, O^t, P^t, E^t_W, E^t_K)$ be a task. We say that operator $o = (N^o, I^o, O^o, P^o, E^o_W, E^o_K)$* matches *task t if $T_{ont} \models N^t(N^o)$ and there is a unique mapping $\sigma = \sigma_1 \cup \sigma_2$ such that mappings $\sigma_1 : I^o \rightarrow I^t$ and $\sigma_2 : O^t \rightarrow O^o$ satisfy the conditions $S \models P^t \sqsubseteq_{\sigma_1} P^o$, $S \models E^o_W \sqsubseteq_{\sigma_1^- \cup \sigma_2} E^t_W$, and $S \models E^o_K \sqsubseteq_{\sigma_1^- \cup \sigma_2} E^t_K$. The matching of a method m with task t is defined similarly.*

Task matching has two steps: First step is, matching based on the task ontology and the second step is, matching based on the preconditions and effects. Task ontology based matching is simply reduced to the problem of instance checking. We have a match if the operator (or the method) is inferred to be an instance of the task in the task ontology T_{ont}.

Matching based on preconditions and effects serves two purposes. First is to ensure that the operator or method selected to achieve the task has the same

constraints about the state transition. For example, if the precondition of a task is satisfied in the current state, the precondition of the matching operator or method should also be satisfied. This means that the precondition of the matching action is allowed to be more relaxed. The relation between the effects of the task and matching action is exactly the opposite. The task description requires us to find an action that will achieve some certain effects. Therefore, the matching action should generate at least the effects specified in the task. Additional effects by the action would be permitted. The subsumption relation between the conditions of the task and the operator (or the method) verifies these constraints.

The second purpose of the matching preconditions and effects is to figure out how the inputs and the outputs of the operator (or the method) align with the inputs and the outputs of the task. Unlike classical HTN formalism, the ordering of the parameters in the task do not have any special meaning. This means that when we have a task with two inputs and a matching operator with also two inputs we do not know which input of the operator corresponds to which input of the task. The containment mapping σ in the query subsumption is used to establish this correspondence between the inputs and the outputs. Note that, the uniqueness of the σ mapping is essential to the definition of task matching. If there are two different mappings that satisfy the subsumption between conditions, it means there is ambiguity about the parameters and we cannot use this action.

Definition 3.13 (State Transition) *Let* $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ *be the current state,* $o = (N, I, O, P, E_W, E_K)$ *be an operator, and* θ *be an assignment of individuals to*

input variables I. Applying operator $o\theta$ in state S causes the current state to change into $S' = \gamma(S, o\theta)$ such that $S' \models E_W\theta \wedge E_K\theta$.

Note that, we have not described the exact semantics of the transition function γ other than saying that state S will change in a way that the effects of the action become true. In the presence of state constraints, defining a precise semantics for such transitions are problematic. As pointed out in [107], some state axioms can be considered as additional *qualifications* for the actions whereas some others encode *ramifications* of the actions. That is, in some cases, the effects of the action might contradict with the axioms in our terminology in a way that we conclude it is not possible to do this action. In other cases, domain axioms tell us how the explicitly asserted effects of an action need to be propagated to produce the implicit effects. But in either case, the terminological component remains same and only the assertional component is modified.

There have been many proposals about update semantics [119, 71, 107, 39, 52] mostly for propositional knowledge bases and without domain axioms. With the abundance of such semantics with different properties and varying advantages and disadvantages, many researchers have argued that there is no single update semantics that can be used in every setting [120].

For this reason, we do not restrict ourselves to one specific update semantics and leave it as a configurable parameter. We adopt the approach of Transaction Logic [11] and say that atomic updates with different semantics can be used in different situations. The only requirement we have from γ function is that effects of

an operator actually hold in the successor state. A reasonable requirement is that the world should change *minimally* as to ensure the effects. That is, a statement holding in a state should persist unless it conflicts with the update. This is very easy to ensure when the state is represented as a set of ground facts but the definition of minimality is quite difficult for the reasons explained above. In Section 3.3.2, we explain the approach currently adopted in HTN-DL algorithm in more detail.

Definition 3.14 (Operator Applicability) *Let* $o = \langle N^o, I^o, O^o, P^o, E^o_W, E^o_K \rangle$ *be an operator,* $S = \langle S_\mathcal{A}, S_\mathcal{T}, S_\mathcal{R} \rangle$ *be the current state,* $t = (N^t, I^t, O^t, P^t, E^t_W, E^t_K)$ *be a task and* θ *be an assignment of individuals to input variables* I^t*. Then we say that operator* o *is applicable to task* t *at state* S *if*

1. *o matches t with mapping σ, and*

2. *the condition $P^o\rho$ is true at state S where ρ is the composite mapping $\sigma\theta$, and*

3. *the resulting state $S' = \gamma(S, o\sigma\theta)$ is consistent.*

We consider the applicability of an operator w.r.t. the inputs that will be used to execute this action. We replace all the appearances of the input variables in the precondition expression with the specified constants. Note that, the precondition expression is not necessarily ground after this substitution because there might be other variables used in the precondition expression.

The operator applicability also requires that the successor state is consistent. The interaction between the facts in the state, state constraints and the update operation γ might cause a contradiction in the final state. Such constraints, called

as *qualification constraints* by Lin and Reiter [80], describe additional qualifications for the actions. Thus, an action violating a state constraint cannot be executed in the real world. While the solution of Lin and Reiter [80] compiles qualification constraints into precondition expressions (and discards the constraints during reasoning about actions) we leave such constraints in the state and verify that they still hold after (simulated) execution of an operator.

Definition 3.15 (Method Applicability) *Let $m = (N^m, I^m, O^m, P^m, V^m, E_W^m, E_K^m, \Gamma)$ be a method where $\Gamma = [(C_1 : \tau_1) \dots (C_n : \tau_n)]$ is a conditional task network, $S = \langle S_{\mathcal{A}}, S_{\mathcal{T}}, S_{\mathcal{R}} \rangle$ be the current state, $t = (N^t, I^t, O^t, P^t, E_W^t, E_K^t)$ be a task, and θ be an assignment of individuals to input variables I^t. A method m is applicable to task t in S if*

1. *m matches t with mapping σ, and*
2. *there exists a mapping ϕ from the variables in V to the individuals in S such that ϕ is in the answer set of $P^m\sigma\theta$ w.r.t. S, and*
3. *there exists i such that $C_i\rho$ is true at state S and $C_j\rho$ is false at state S $\forall j < i$ where the mapping ρ is defined as $\sigma\theta \cup \phi$.*

We will say that τ_i is the active task network *in Γ. $\tau_i\rho$, the task network obtained by applying substitution ρ to the active network, is called a* simple reduction *of t by m in S.*

Note that, due to incomplete knowledge, there might be cases where the method is not applicable although its precondition is satisfied. This is because

the conditional task network describes a nested *if-then-else* structure and for an *if-then-else* statement one or the other choice should be selected based on the truth value of the condition. If the truth value is unknown, as might be the case, the planner has no way of choosing between two possibilities. Choosing one option over the other might cause the planner to generate unsound plans (these are unsound plans w.r.t. complete knowledge of the world). For this reason, to avoid ambiguity, we say a method is applicable only if all the conditions can be proved or disproved.

This requirement is too restrictive in the sense that we completely give up when we realize we do not have enough information. An alternative would be to generate conditional plans based on these unknown conditions in the hope that as we start executing the actions in the plan we might gather the necessary information to make a decision.

Next we are going to formally define task decomposition. The definition is very similar to the one given in [37] thus it is very complicated. Intuitively what it says is given a state S and a task u with no predecessors in a task network w we can replace the task u with a simple reduction of u by a method in S. For this, we need to update the nodes, the edges and the parameter bindings to change any references of u to the subtasks in the reduction. Also as discussed in [37] when dealing with partially ordered tasks we need to enforce the preconditions of m until at least one subtask is accomplished. That is why task decomposition definition returns a set of task networks, i.e. one for each subtask that has no predecessor in the reduction.

Definition 3.16 (Task Decomposition) *Let S be a state, $w = (\eta, \epsilon, \lambda)$ be a task*

network, $u \in \eta$ be a node that has no predecessors. Let $succ(u)$ be the set of all immediate successors of u. Also let $w' = (\eta', \epsilon', \lambda')$ be the task network obtained by removing the node u, the edges and the parameter bindings involving u from w. Suppose task network $w_m = (\eta_m, \epsilon_m, \lambda_m)$ is a simple reduction of t_u by a method m in S with θ where θ is a mapping for input parameters of t_u based on the parameter bindings in λ. Let $start(w_m)$ be the set of all nodes in η_m that has no predecessors. If η_m is nonempty, then the result of decomposing u in w by w_m (denoted by $\delta(u, w, w_m)$) is a set of task networks $\{(\eta' \cup \eta_m, \epsilon' \cup \epsilon_v, \lambda^*) \mid v \in start(w_m)\}$ where

- $\epsilon_v = \epsilon_m \cup \{(g, h) \mid \ g \in \eta_m,\ h \in succ(u)\} \cup \{(v, u') \mid u' \in start(w) - \{u\}\}$
- $\lambda^* = \lambda' \cup \{a \leftarrow b \mid a \leftarrow \langle u, p\rangle \in \lambda \text{ and } p \leftarrow b \in \lambda_m\} \cup \lambda'_m$ *where*

  $\lambda'_m = \lambda_m - \{a \leftarrow b \mid a \leftarrow b \in \lambda_m \text{ and } a \in O^{t_u}\}$

  If η_m is empty then $\delta(u, w, w_m) = \{(\eta', \epsilon', \lambda^*)\}$.

Finally, we define when a plan is considered to be a solution for a planning problem.

Definition 3.17 (Plan) *Let $P = (S_0, w, D)$ be a planning problem. The plan $\pi = \langle o_1, o_2, \ldots, o_n\rangle$ is a solution for P if one of the following conditions hold:*

- *Both w and π are empty.*
- *There is a task node $\langle l_u : t_u\rangle \in w$ and there is no $\langle l_v : t_v\rangle \in w$ such that $l_v \prec l_u$, and*
  - *Operator o_1 applicable to t_u in S_0 and the plan $\pi' = \langle o_2, \ldots, o_n\rangle$ is a solution for the planning problem $P' = (\gamma(S_o, o_1), w \setminus \{u\}, D)$, or*

– *There is a simple reduction w_m of t_u by a method m in S_0 and π is a solution for $P' = (S_0, w', d)$ where $w' \in \delta(u, w, w_m)$.*

The set of all solutions to a planning problem is denoted by solves(S_0, P, D).

3.3 HTN-DL Algorithm

In this section, we present the HTN-DL algorithm and explain the details of how condition evaluation and state updates are done. Algorithm 1 shows the pseudo-code of the planning algorithm. This is a forward-decomposition HTN planning algorithm working in the same spirit as other forward-planning HTN systems such as SHOP2 [94]. The biggest difference in HTN-DL is line 5 where the `find-applicable` procedure is called to find operators and methods applicable to the selected task in the current state. Note that, we do not distinguish between primitive and non-primitive tasks and we can match both an operator and a method with the same task.

Algorithm 2 shows a simplified version of the `find-applicable` procedure[3]. We iterate through the instances of the given task and find the matching operators and methods. We check for applicability as defined in the HTN-DL semantics. For operators, we check the satisfiability of the precondition and ensure the consistency of the successor state. For methods, we again check for the satisfiability of the precondition and then find the active task network. Note that, as discussed in the

[3]For brevity, we do not include the parts related to variable bindings, e.g. normally we apply the input bindings to the precondition expression before we check for satisfiability and evaluating method precondition involves retrieving bindings for the local variables

Algorithm 1 HTN-DL(S, w, D)

```
1:  if w is empty then
2:      return ⟨⟩                  // an empty plan is returned for empty task network
3:  end if
4:  Let u = ⟨l : t⟩ be a node in w with no predecessors
5:  Let A = find-applicable(S, t, D) be the set of all applicable operators
        and methods
6:  if A is empty then
7:      return fail
8:  end if
9:  Nondeterministically choose an action from A
10: if an operator o is chosen then
11:     Let S′ = γ(S, o)
12:     Let w′ = w \ {u}
13:     return ⟨o, HTN-DL(S’,w’,D)⟩
14: else if a method m is chosen then
15:     Let w_m be a simple reduction of t by m in S
16:     Nondeterministically choose a w′ from δ(u, w, w_m)
17:     return HTN-DL(S, w′, D)
18: end if
```

previous section, a method will be considered inapplicable if a condition can be neither proved or disproved.

3.3.1 Evaluating Conditions

The problem of evaluating conditional expressions is directly reducible to DL query answering. The preconditions of operators and the conditions inside a conditional task network are boolean queries that require a `true`/`false` answer. The preconditions of methods, on the other hand, are retrieval queries where the local variables of a method are the distinguished variables of that query. We will examine

Algorithm 2 find-applicable(S, t, D)

Inputs: S is the current state, $t = (N^t, I^t, O^t, P^t, E^t_W, E^t_K)$ is the task,
$D = (O, M, T_{ont})$ is the planning domain

1: Let $result = \{\}$ *// result is the set of matching operators and methods*
2: **for all** a such that $T_{ont} \models N^t(a)$ **do**
3: **if** $o \in O$ such that $o = (a, I^o, O^o, P^o, E^o_W, E^o_K)$ **then**
4: *// a is an operator name*
5: **if** $S \not\models P^o$ **then** continue loop **end if**
6: **if** $S \models P^t \sqsubseteq P^o$ and $S \models E^o_W \sqsubseteq E^t_W$ and $S \models E^o_K \sqsubseteq E^t_K$ **then**
7: Let $S' = \gamma(S, o)$
8: **if** S' is consistent **then**
9: Let $result = result \cup \{o\}$
10: **end if**
11: **end if**
12: **else if** $m \in M$ such that $m = (a, I^m, O^m, V^m, P^m, E^m_W, E^m_K, \Gamma)$ **then**
13: *// a is a method name*
14: **if** $S \not\models P^o$ **then** continue loop **end if**
15: **if** $S \models P^t \sqsubseteq P^m$ and $S \models E^m_W \sqsubseteq E^t_W$ and $S \models E^m_K \sqsubseteq E^t_K$ **then**
16: Let n be the number of task networks in Γ
17: **for** $i = 1$ to n **do** *// this loop finds the active task network*
18: **if** $S \models C_i$ **then**
19: Let $result = result \cup \{m\}$
20: **else if** $S \models \neg C_i$ **then**
21: **continue loop**
22: **end if**
23: *// either we found the active network or C_i can be neither proved nor disproved. In either case we are done with this method*
24: **exit loop**
25: **end for**
26: **end if**
27: **end if**
28: **end for**
29: **return** result

both cases separately.

Answering boolean conjunctive queries can be reduced to KB consistency checking using the so-called "rolling-up" technique [58, 64]. This technique works by creating a concept expression from the conjunctive query and checking if that concept has a non-empty interpretation in every model of the KB. If $\mathcal{K}$ is the original KB, Q is the boolean conjunctive query, C_Q is the concept we obtain by rolling-up Q then we say $\mathcal{K} \models Q$ iff $K' = K \cup \{\top \sqsubseteq \neg C_Q\}$ is inconsistent.

There are some limitations of the rolling-up technique based on the expressivity of the KB and the features of the query. We treat the conjunctive query as a graph where each variable is a node and each role is a directed edge between nodes. If there are constants (individuals) used in the query atoms, we can simply get rid of such atoms using one step of rolling-up. For example, a query atom $p(x, \mathtt{a})$ (resp. $p(\mathtt{a}, x)$) where p is a role, x is a variable, and $\mathtt{a}$ is a constant, can be turned into $(\exists p.\{a\})(x)$ (resp. $(\exists p^{-}.\{a\})(x)$).

Without loss of generality, we can assume that the query graph is weakly connected; that is, there is an *undirected* path between any two nodes in the graph. We can split a disconnected query graph into weakly connected, mutually disjoint subgraphs. The original query is satisfied by the KB if all the subqueries are satisfied. for this reason, we will only consider weakly connected graphs.

The rolling-up technique can be used for answering boolean conjunctive queries if there are no cycles in the query graph. Recall that, all the variables in a boolean query are non-distinguished (we will later discuss the case of distinguished variables separately). If the KB does not contain any transitive and inverse roles, e.g. the

DL expressivity $\mathcal{ALCN}$, then the cycles in the query can only be caused by existing individuals (due to the tree-model property of the language). In that case, we can turn the query into a retrieval query by treating the variables in the cycle as distinguished variables. The boolean query should be `true` if and only if the answer set of the retrieval query is non empty.

However, if the KB in question uses the full expressivity of OWL-DL, i.e. the DL $\mathcal{SHOIN}$, this approach is not applicable. For this reason, we will restrict the precondition expressions to non-cyclic queries. This is actually not a serious problem in practice for the following reason: The condition expressions of HTN-DL operators and methods nearly always refer to input variables. During planning, we substitute input variables with the given input bindings before we evaluate the condition. Therefore, even though the initial query expression is cyclic, the query we actually evaluate may not contain any cycles as the variables causing the problem are replaced with constant values.

For task decomposition, we also need to decide when a negation of the query is entailed. As we explained above, if $\mathcal{K}$ is a KB, Q is a boolean conjunctive query, C_Q is the concept we obtain by rolling-up Q then $\mathcal{K} \models Q$ iff $K' = K \cup \{\top \sqsubseteq \neg C_Q\}$ is inconsistent. If $\mathcal{K}'$ is consistent then all we can conclude is $\mathcal{K} \not\models Q$ which is not equivalent to $K \models \neg Q$. The condition $K \not\models Q$ is weaker because it just says that it is possible but not necessary that Q is false in the world. If we plan some actions on this possibility, we might end up generating unsound plans. For this reason, in HTN-DL semantics, finding the active network in a conditional task networks is done by checking the entailment of negated query, i.e. checking $K \models \neg Q$.

If we want to check that the negation of a query Q is entailed then we can again use rolling-up to obtain C_Q and test its (un)satisfiability. C_Q is satisfiable only if there is a model $\mathcal{I} \models \mathcal{K}$ s.t. $C_Q^{\mathcal{I}} \neq \emptyset$. If such a model exists, it is straight-forward to show that every non-distinguished variable in the query can be mapped to an element of $\Delta^{\mathcal{I}}$ in this interpretation satisfying every query atom. Therefore, we can say that $\mathcal{K} \models \neg Q$ iff C_Q is unsatisfiable w.r.t. $\mathcal{K}$.

Answering retrieval queries can simply be reduced to boolean query answering by substituting the distinguished variables in the original query with the individuals from KB. If the resulting query (which might still have non-distinguished variables in it) is entailed by the KB then the substitution is added to the answer set. If the original query contains a cycle with only non-distinguished variables then we would not be able to answer the query. If the cycle has at least one distinguished variable (whose removal breaks the cycle) then we would still be able to roll-up the partially-grounded query (since constants do not cause cycles). Although, this approach is technically sound, it is not practical. We will address this issue in detail in Chapter 6.

3.3.2 Updating State

During planning, the planner will simulate the effects of actions as operators are added to the plan. The state will be updated according to the effects of the action. The physical effects of an action describe what will change in the world. Using this description planner needs to update its local state such that this local

state will still represent the world state correctly. We will now describe how to handle world-altering effects and knowledge effects separately.

As we explained in Section 3.2.3, defining the semantics of an update operation in an expressive KR language is not straight-forward. This is an extensively studied research topic [119, 71, 107, 39, 52] in reasoning about action community but there is not a consensus on a single correct solution even for propositional KBs. The postulates put forward by Katsuno and Mendelzon [71] proposes some basic characteristics an update operation should have but the update semantics developed in the literature do not agree on these basic properties as examined in detail by Herzig and Rifi [52].

Currently the approach we adopt for state updates is a formula-based approach close to WIDTIO (When In Doubt Throw it Out [119]) approach [38, 39]. According to this update operator, updating the KB $\mathcal{K}$ with $\mathcal{U}$ yields $\mathcal{K}' = \mathcal{K} \cup \mathcal{U}$ if $\mathcal{K}'$ is consistent. If there is an inconsistency a minimal number of assertions are removed from $\mathcal{K}$ such that addition of $\mathcal{U}$ will not cause an inconsistency. The state constraints are protected from the removal operation. The implementation of formula-based updates is very straight-forward using the axiom pinpointing service [70] developed for $\mathcal{SHOIN}$.

Although, formula-based update operation is computationally attractive, it also has disadvantages. With WIDTIO approach, there is a danger of retracting too much information in order to eliminate inconsistency thus leading to less and less knowledge as planning continues. Since our removal operation is constrained to ABox assertions the effects of this problem is somewhat limited. The main reason

for adopting this update operation was to reuse existing efficient implementation for updates. As we mentioned in the previous section, it would be fine to use a different update operation such as the recently proposed model-based update operations for DLs [81, 4].

When a service has only knowledge effects we can simply execute this service and add the information to the KB (we will discuss this in more detail in the next section). If the action has both world-altering effects and knowledge effects then execution is not desirable without ensuring the subsequent steps in the plan will be successful. Since knowledge effects most typically refer to the outputs of the service, it is not possible to know what the exact effects of the service will be without execution. In such cases, we assign a skolem constant to each output variable, apply this substitution to the effect description to obtain a ground set of ABox assertions and add these assertions to the state.

Note that, being cautious and not executing any actions with world-altering effects has some consequences. In some cases, we will not be able to find some plans because we do not know what the outcome of execution will be. For example, consider the $studentRegistration$ and $notify$ tasks used in Example 3.3. The task notify uses the output from the task $studentRegistration$ but during planning we will not know the value of the output $transactionID$. In this example, we can still generate a plan because all we have to verify is that $transactionID$ should be of type $Transaction$ which is guaranteed by the task definition. But if $notify$ task had to make another decision based on the value of $transactionID$ then plan would fail. This is because there is no information about the skolem constant we

introduced for output *transactionID* other than what is stated in the effects of *studentRegistration* so nothing else can be proved from the planner's local state.

It is possible to relax the restriction about execution of world-altering services. For example, suppose we execute *studentRegistration* service but later fail to find a plan (e.g. the billing service cannot be executed). We could backtrack from this situation if there is a service *dropClass* that lets us to remove the course from student's schedule. If there is an *inverse action* for the service we are interested in then we can safely execute this service and use the inverse action for *undoing* the effects. For example, such a backtracking strategy has been adopted in the PUCCINI planner [40].

3.3.3 Interleaving Execution and Planning

As we have previously discussed, executing purely information-providing services during planning does not have any undesired side-effects and is essential to solve many real world problems. The information returned from the service will be about the initial state of the planning problem. Since none of the world-altering actions planner simulated so far has been executed in the real world. For this reason, executing information-providing services during planning should be conceptually equivalent to executing them before planning started.

Executing information providing services in the initial state is straight-forward. We can simply add the information returned by the service to the local state. We do not need to use the update function to apply the knowledge effects because neither

the actual state of the world nor the planner's state has changed.

The situation is different if the information is gathered at a point where the planner already modified its local state. Adding the retrieved information to the intermediate state could cause us to generate invalid plans. For example, consider the case where the planner execute a Web Service to get the available appointment times from a hospital and then planner simulates scheduling an appointment at one of the available time slots. If the information-providing service is executed again, the service would return the exact same available appointment times since the appointment has not been done in the real world. Adding the same available times to the state would be a problem because planner would be able to schedule another appointment in the same time slot.

If the information-providing services were to return only boolean answers, e.g. given a ground conjunctive query the service returns `true` or `false`, then we could use the solution described in [105] which is to *guard* the information-providing actions: Do not execute the sensing action if the truth value of the query is known. This simple criteria is enough to ensure that the planner will not overwrite the planned actions with information gathered. However, in HTN-DL setting, information-providing services return a set of values that make the knowledge effects `true` and we cannot adopt this solution. Requiring that no information-providing service will be executed more than once is also not satisfactory as two different services might return overlapping information. In the most general case, the safe solution is to add the information to initial state and apply the effects of partial plan generated so far. If the update operation ensures the condition

$\gamma(K_1, U) \cup \gamma(K_2, U) = \gamma(K_1 \cup K_2, U)$ then we can simply apply the effects to the gathered information and combine the results with the current intermediate state.

Chapter 4

Translating Web Service Descriptions to HTN-DL

This chapter discusses how HTN-DL can be used for Web Service composition problems. We will first discuss how HTN-DL relates to Web Services in general and then examine the relation between HTN-DL and OWL-S. We specifically focus on OWL-S language as it is currently the most mature and the most widely deployed Semantic Web Services technology.

We present a translation algorithm that will generate HTN-DL domains from OWL-S descriptions. We discuss translation of profile descriptions and process models separately and examine each control construct in detail. This translation shows that the control flow of a Web Service workflow can be encoded using HTN-DL methods. The algorithms presented in this chapter can be used to represent the control constructs in other Web Service description languages.

The translation algorithm also provides a formal semantics to OWL-S language. OWL-S specification does not have a formal semantics but one has been given by Narayanan and McIlraith [92] in terms of the situation calculus [106] and Golog [77]. We show that both semantics are equivalent for the subset they both cover. In addition, HTN-DL can encode additional constructs, such as partially ordered composite services that interleave, and provides a additional features such as services that are both world-altering and information-providing.

4.1 Relation between OWL-S and HTN-DL

As we have reviewed in Section 2.4, OWL-S language partitions a service description into three components: service profile; process model and grounding. Service profile is used to describe the service capabilities defined through inputs, outputs, preconditions and effects as well as attributes such as the quality of service provided, security guarantees made, etc. Process model describes the pattern of interaction with the Web Service. Grounding defines the execution details of the Web Service by linking the process definition typically to a WSDL operation (other kind of groundings are also possible).

OWL-S process ontology is a quite extensive orchestration language that provides control flow elements such as `Perform`, `Sequence`, `Any-Order`, `Choice`, `Split`, `If-Then-Else`, `Repeat-While`, and `Repeat-Until`. A CompositeProcess in OWL-S can be built from other processes using these constructs. Processes that do not have any internal structure (or whose internals are hidden) are defined as AtomicProcesses. Roughly, an OWL-S AtomicProcess corresponds to an HTN-DL operator and an OWL-S CompositeProcess corresponds to an HTN-DL method.

OWL-S process models can express a number of workflow patterns characterized in the literature [121, 117]. These pre-defined compositions may integrate services from different locations and they are executable. However, every step defined in the workflow is bound to one concrete process. For this reason, such composite process descriptions are not flexible enough to be used in different situations. In the event that one of the component services is not available anymore the whole

composite process description would fail.

What is missing in OWL-S process ontology is a way of describing abstract functionality that can be matched with concrete services at run-time or plan-time. OWL-S defines a third type of processes named SimpleProcess which was originally envisaged to address a similar problem but this construct is still under-specified as of the latest release (version 1.1). Technical overview states that SimpleProcess is primarily designed to provide an abstract view for an existing atomic or composite process. That is, it is mainly used when the concrete process that corresponds to the abstract view is known a priori at design-time.

In order to make OWL-S capable of representing abstract processes we need something similar to an HTN-DL task. Abstract processes should not be tied to any specific service but should describe a functionality that we can match with the capabilities of existing services. Such a functionality description should certainly refer to profile descriptions. The non-functional attributes described in the profile section are required to differentiate the candidate services so that best possible choice can be found.

In order to describe such abstract processes, we propose to add a new process type AbstractProcess to OWL-S process ontology. Equivalently, the specification of SimpleProcess can be cleared up to serve such a purpose. To avoid ambiguity we will use the name AbstractProcess throughout this chapter.

An AbstractProcess is similar to an AtomicProcess description and has inputs, outputs, preconditions and effects. The difference is that AbstractProcesses are not directly connected to any specific *Profile* or *Grounding* description. And unlike

SimpleProcess, it does not have any links to existing processes. AbstractProcess being another type of *Process*, can be used inside a *Perform* construct. This allows us to have *partially abstract* templates, e.g. we can have a sequence where first step is a concrete AtomicProcess that is executable and second step is an AbstractProcess that needs to be instantiated.

The following is an example that describes an *AbstractProcess* that sells books and accepts credit cards Visa or Mastercard[1].

```
(define abstract process GetBook
  inputs: ( RequestedBook - books:Book,
            ShipmentAdress - loc:Address,
            Payment - :VisaOrMastercard
  outputs:( Receipt - :PurchaseReceipt )
  result: (
     shippedTo(RequestedBook, ShipmentAdress) )
);
```

When we want to perform an abstract process we would express this with a *Perform* construct. As an additional parameter to *Perform* we would specify what kind of service profiles would be acceptable in matching. The following example shows a *Perform* statement where the matching services for *GetBook* are limited to services that sell new or used books and has high user rating. Services that lend books (e.g. libraries and book clubs) are explicitly disallowed in the profile description.

```
:PerformBookBuy a p:Perform ;
    p:process  :GetBook ;
```

[1]For brevity, process descriptions are written in OWL-S presentation syntax and all other ontological definitions are given in N3 syntax. Both formats can easily be translated to verbose RDF/XML syntax.

```
p:profile [
  a owl:Class;
  [ owl:intersectionOf (
     [ owl:onProperty :hasUserRating;
       owl:someValuesFrom :High ]
     [ owl:unionOf (
        :NewBookBuyingService
        :UsedBookBuyingService ) ]
     [ owl:complementOf
        :BorrowBookService ] ) ] ] .
```

4.2 From OWL-S to HTN-DL

In this section, we explain how OWL-S service descriptions can be encoded as HTN-DL domains. For each service, the profile description is translated to an element in the task ontology and the process model is translated to a set of methods and operators.

Before we present the translation algorithm, we first discuss some of the differences between OWL-S and HTN-DL representations and explain how this effects the translation algorithm. Then we describe how to translate profile descriptions and process models separately.

4.2.1 Translating Profile Descriptions

The profile descriptions in OWL-S are represented as a set of ABox assertions. The profile of a service is represented as an instance of a specific concept named *ServiceProfile*. The profile descriptions are associated with "service parameters" that describe different features of the service. The service parameters are simply

some relations asserted in the ABox. These relations may be about many different things such as the quality of service, e.g. the average response time, the security protocol, e.g. encryption algorithm, the service provider, e.g. where the provider is located and what credentials they have.

The ontologies used to describe OWL-S services might also describe profile hierarchies. Profile hierarchy is a categorization of services where typically the *ServiceProfile* is the top concept. Categories of services are defined by describing some restrictions on the service parameters of the profile.

Since OWL-S profile descriptions are nothing more than OWL ontologies we can directly use them as HTN-DL task ontologies. The only issue here is that the individuals in the task ontology should be mapped to operator and method definitions we have. In the next section, we will show how operator and method descriptions are generated from process models. Therefore, all we need to do is relate the operators and methods coming from translation of process models to the individuals coming from the translation of profile descriptions. As OWL-S service description tells us the profile and process belonging to the service, establishing this relation is also trivial.

4.2.2 Translating Process Models

There are some representational and expressivity differences between OWL-S and HTN-DL that make an exact translation impossible. One major difference was the lack of abstract processes in OWL-S which we have already discussed. Next, we

will point out some other issues.

The effects of an OWL-S process is not described in two categories as in HTN-DL. Therefore, there is no way to tell whether a service has any world-altering effects or it is just an information-providing service. This problem can be solved easily if we annotate the (appropriate) OWL-S effects as knowledge effects as we do in HTN-DL.

OWL-S allows one to express *conditional effects*. A conditional effect says that the change in the world will happen only if a certain condition is met. This is different from the precondition description because the result of executing of an action in a state where precondition is not satisfied have an undefined effect. Conditional effects, on the other hand, describe the effects precisely but under different conditions different effects might occur. Although, HTN-DL does not allow conditional effects for operators or methods, one can simulate this expressivity using an additional method. For example, if an operator has a conditional effect, we could define a method and use the conditions of the effects as the conditions of the task network. At each task network, we would use a single method that is achieved by a slightly modified version of the operator, i.e. augmented with corresponding effect description. To keep the presentation simple, we will only focus on unconditional effects in our translation algorithm.

One feature of OWL-S not supported in HTN-DL is concurrent processes. The control constructs `Split` and `Split-Join` describe a set of process that can be executed concurrently with or without synchronization, respectively. In HTN-DL, we do not have concurrent actions but we allow interleaving of composite actions. Using this feature, we choose to translate `Split` and `Split+Join` as unordered set of

methods that can be interleaved arbitrarily. Although this encoding will rule out the plans where two atomic processes are executed concurrently, the plan still is correct with respect to the semantics because there is no hard constraint for simultaneous execution.

There are also some areas where OWL-S specification is not clear. For example, using the output of one process invocation as an input for another process invocation is common practice. But doing so asserts an implicit ordering relation between two invocations. The OWL-S specification does not particularly say what happens when such a data flow link is defined between two performs that are used in the same `Split-Join` construct. To resolve this ambiguity, we will assume that such data flow links do not exist in the process models we are translating to HTN-DL.

We are now ready to present the translation algorithm. We will start with the translation of logical conditions, then examine each control construct separately and finally wrap up with the translation of processes.

Translating Conditions

OWL-S allows logical expressions to be expressed in several different languages such as Semantic Web Rule Language (SWRL) [59], Knowledge Interchange Format (KIF) [35], and Declarative RDF System (DRS) [85, 86]. These languages have different expressive power to describe different kind of formulas but generally the logical expressions in OWL-S are limited to conjunctions of (possibly nonground) OWL facts. For example, if SWRL is being used to express a condition then the

expression is simply a list of atoms (named as AtomList) but not a complete rule definition. For the purposes of the translation, we will assume SWRL syntax is being used.

SWRL atom lists can be directly expressed as HTN-DL conditions since both expressions are in the same form. One additional atom type in SWRL is *built-in atoms*. SWRL built-in atoms include operations for comparing data values, performing mathematical computation, manipulating strings, etc. Such operations are not included in the HTN-DL formalism but it is very straight-forward to incorporate such extensions to HTN-DL in the style of SHOP2 since HTN-DL is also a forward-planning algorithm. Evaluating conditions in a state yields values for variables to which we can apply these built-in functions.

Translating Control Constructs

We translate each OWL-S control construct to an HTN-DL task and a set of HTN-DL methods achieving that task. Translating each construct requires slightly different operations as we will present in this section. Algorithm 3 shows the pseudo-code of the generic control construct translation function that simply calls the corresponding translation function.

When we are translating the control constructs to HTN-DL we not only need to have the same control flow but also have the same data flow. There are two different places in OWL-S process models to specify data flow. First is when we are *perform*ing a process we specify where the inputs are coming from; which is either

Algorithm 3 *Translate-Construct*(X)

Inputs: X is an OWL-S control construct, P is the enclosing parent process

Outputs: Tuple $\langle t, D\rangle$ where t is a task description for X and D is the domain description containing all the operators and methods generated for X

```
1:  if X is a Sequence construct then
2:      return Translate-Sequence(X)
3:  else if X is a Choice construct then
4:      return Translate-Choice(X)
5:  else if X is a Any-Order construct then
6:      return Translate-Any-Order(X)
7:  else if X is a Split-Join construct then
8:      return Translate-Split-Join(X)
9:  else if X is a If-Then-Else construct then
10:     return Translate-If-Then-Else(X)
11: else if X is a Repeat-While construct then
12:     return Translate-Repeat-While(X)
13: else if X is a Repeat-Until construct then
14:     return Translate-Repeat-Until(X)
15: else if X is a Perform construct then
16:     return Translate-Perform(X)
17: end if
```

the input (or a local variable) of the enclosing composite process or the output of a previous perform statement. Second is where we specify how the output of composite service is linked to an output of a perform statement.

In theory, the data flow specification of OWL-S can be directly expressed using the parameter bindings in a task network description. However, due to nesting of control constructs, two `Perform` statements sharing a variable might end up in different method descriptions. For this reason, we need to carry these variables through the inputs and outputs of methods we generate.

In order to handle the data flow, we first define the inputs and outputs associ-

ated with a control construct. The *scope* of a control construct is defined recursively as the control construct itself union the scope of all its components. A control construct X *produces* a variable v if v is the output of a perform statement in the scope of X. A control construct X *uses* a variable v if v is used as input to a perform statement in the scope of X or it is used in a logical condition in the scope of X. A control construct X *requires* a variable v if X uses v but does not produce v. The inputs of a control construct X (denoted as $In(X)$) is all the variables X requires. The outputs of a control construct X (denoted as $Out(X)$) is all the variables X produces. Computation of $In(X)$ and $Out(X)$ is done trivially by traversing the control constructs recursively.

Given these definitions, the data flow between nested control constructs is straight-forward to write. Suppose, construct Y is a component of the construct X, Then for each parameter in $In(Y)$ and $Out(Y)$, we need a parameter binding to the corresponding variable in the construct X. In a `Sequence` construct, a construct might use the output of a sibling, so we need to have additional bindings for those cases. These are standard data flow elements that need to exist in every construct. We will call these standard data flow bindings $DF(X)$ and directly use in our translation. In some cases, namely repeat loops, we will actually need some additional bindings which we will explicitly describe in the translation procedure.

There is still one more issue we need to solve. As we mentioned at the beginning of this section, for each control construct, we generate one task and one or more methods that can achieve that task. The task ontology tells us exactly which methods we can use to achieve the task. But the mapping between the parameters

of the task and the parameters of the method need to be established through the precondition and effect expressions. Note that, even though we use the same variable names for the parameters of the method and the task, the planner would not still match the parameters because the same variable names in different contexts are treated as coincidental.

The problem with parameter mapping in this case is that control constructs do not have preconditions or effects; only processes have preconditions and effects. Therefore, we need to come up with such precondition and effects expressions that they will enable us to match the parameters but these additional expressions will not affect the planning process. For this purpose, we will create additional concept names $In_1, In_2, In_3, \ldots, Out_1, Out_2, Out_3, \ldots$ and use them in precondition and effect expressions. If a control construct X has two inputs $In(X) = \{y, z\}$ then the precondition expression will be $Pre(X) = In_1(y) \wedge In_2(z)$. The effect expression $Eff(X)$ will be computed similarly. We will use the same precondition and effect expression in the task and the method we generate so that parameters will be matched.

We are finally ready to describe the translation procedure for specific control constructs. Each control construct is examined separately with the exception of `Produce` construct. `Produce` construct is syntactic sugar for describing the parameter bindings for the outputs of a composite process. They do not describe any additional functionality and only affect the data flow bindings. Therefore, `Produce` constructs will be processed by the $DF(X)$ procedure and we will omit them in the following translation functions.

Sequence Translation of a `Sequence` construct is quite straight-forward. We recursively translate all the components of the sequence and create one task network where the tasks are totally ordered.

Algorithm 4 *Translate-Sequence*(X)

Inputs: X is a `Sequence` construct in the form X_1 ; X_2 ; ... ; X_k

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain

2: Let $t = (N_t, \mathit{In}(X), \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset)$ be a task description where N_t is a unique task name

3: **for all** $i : 1 \leq i \leq k$ **do**

4: Let $\langle t_i, D_i \rangle = \mathit{Translate\text{-}Construct}(X_i)$

5: Let n_i be a task node for task t_i

6: **end for**

7: Let $m = (N_m, \mathit{In}(X), \emptyset, \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset,$
$[\top : (\{n_1, \ldots, n_k\}, \{(n_i, n_{i+1}) \mid 1 \leq i < k\}, \mathit{DF}(X))])$

8: Add method m to M

9: Add a new concept N_t, a new individual N_m and assertion $N_t(N_m)$ to T_{ont}

10: **return** $\langle t, D \cup \bigcup_{1 \leq i \leq k} D_i \rangle$

Note that, translating component constructs could create more elements in the HTN-DL domain so we combine the results of recursive steps in the end. The complete translation code for `Sequence` is shown in Algorithm 4.

Choice Executing a `Choice` control construct is achieved by executing any one of its components. There is no requirement other than the chosen construct being executable, i.e. the preconditions of all the involved processes are satisfied. The semantics of this construct is very similar to the relation between HTN-DL tasks and methods. a task can be achieved by many different methods as long as the method (and all its subtasks) are applicable in the given state. Using this similarity,

Algorithm 5 *Translate-Choice*(X)

Inputs: X is a `Choice` construct in the form: X_1 ;? X_2 ;? ... ;? X_k

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain
2: Let $t = (N_t, \mathit{In}(X), \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset)$ be a task description where N_t is a unique task name
3: **for all** $i : 1 \leq i \leq k$ **do**
4: Let $\langle t_i, D_i \rangle = \mathit{Translate\text{-}Construct}(X_i)$
5: Let n_i be a task node for task t_i
6: Let $m_i = (N_{m_i}, \mathit{In}(X), \emptyset, \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset,$ $[\top : (\{n_i\}, \emptyset, DF(X))])$
7: Add method m_i to M
8: Add assertion $N_t(N_{m_i})$ to T_{ont}
9: **end for**
10: **return** $\langle t, D \cup \bigcup_{1 \leq i \leq k} D_i \rangle$

we translate a `Choice` construct into one task and a set of methods (one method corresponding to each component) as shown in algorithm 5.

Any-Order The construct `Any-Order` allows its components to be executed in arbitrary order but disallows any interleaving between components. Therefore, it is merely syntactic sugar for defining a choice over all possible permutations of the component constructs. For this reason, we will omit the translation of this construct.

Split-Join As we mentioned earlier, we translate `Split-Join` constructs as possibly interleaving non-concurrent task networks. For this reason, the translation of a `Split-Join` is quite similar to `Sequence` translation the only difference being that there are are no edges (i.e. ordering constraints) in the generated task network. The Algorithm 6 shows the translation function.

Algorithm 6 *Translate-Split-Join*(X)

Inputs: X is a `Split-Join` construct in the form: $X_1 \,||\!>\, X_2 \,||\!>\, \ldots \,||\!>\, X_k$

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain
2: Let $t = (N_t, \mathit{In}(X), \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset)$ be a task description where N_t is a unique task name
3: **for all** $i : 1 \leq i \leq k$ **do**
4: Let $\langle t_i, D_i \rangle = \mathit{Translate\text{-}Construct}(X_i)$
5: Let n_i be a task node for task t_i
6: **end for**
7: Let $m = (N_m, \mathit{In}(X), \emptyset, \mathit{Out}(X), \mathit{Pre}(X), \mathit{Eff}(X), \emptyset,$
$[\top : (\{n_1, \ldots, n_k\}, \emptyset, \mathit{DF}(X))])$
8: Add method m to M
9: Add a new concept N_t, a new individual N_m and assertion $N_t(N_m)$ to T_{ont}
10: **return** $\langle t, D \cup \bigcup_{1 \leq i \leq k} D_i \rangle$

If-Then-Else The `If-Then-Else` construct defines a simple conditional statement. If the condition specified is satisfied in the current state the construct in the "then" part is executed. Otherwise, if there is an "else" part we execute the construct specified there. Note that, the condition of the `If-Then-Else` is not like a precondition. The unsatisfiability of a precondition indicates a failure whereas it is all right when the "if" condition is false. We simply do not perform the action. For this reason, we use the conditional task networks to translate `If-Then-Else` structures rather than precondition expressions. Algorithm 7 shows the translation procedure for the case where there is an "else" component. In the case, where the "else" part is missing, we can simply omit n_2 and use an empty task network for the second network.

Algorithm 7 *Translate-If-Then-Else*(X)

Inputs: X is in the form: `if(` $Cond$ `)` `then` X_1 `else` X_2

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain
2: Let $t = (N_t, In(X), Out(X), Pre(X), Eff(X), \emptyset)$ be a task description where N_t is a unique task name
3: **for all** $i : 1 \leq i \leq 2$ **do**
4: Let $\langle t_i, D_i \rangle = $ *Translate-Construct*(X_i)
5: Let n_i be a task node for task t_i
6: **end for**
7: Let $m = (N_m, In(X), \emptyset, Out(X), Pre(X), Eff(X), \emptyset,$
$[\ Cond : (\{n_1\}, \emptyset, DF(X))$
$\top : (\{n_2\}, \emptyset, DF(X))])$
8: Add method m to M
9: Add a new concept N_t, a new individual N_m and assertion $N_t(N_m)$ to T_{ont}
10: **return** $\langle t, D \cup D_1 \cup D_2 \rangle$

Repeat-While The construct `Repeat-While` defines a structure where a control construct is repeated as long as a certain condition holds in the state. We can encode such a loop using recursive task networks, that is, a method has a subtask which is the same task that method achieves.

Although the main idea behind this translation is simple, the details are a little complicated. Basically, we are creating one method that has a conditional task network with two task networks. The first task network (which has the condition used in the `Repeat-While` construct) first achieves the task corresponding to the body of the repeat loop and then recursively applies the same method again. When the condition becomes false at a certain step, we use the second task network which is simply empty. This second empty task network is needed to distinguish between a failure, i.e. a task not being achieved, and the case where the loop condition becomes false. This task network is empty so applying this task network will not

affect the resulting plan or the state of the world.

Algorithm 8 *Translate-Repeat-While*(X)

Inputs: X is in the form: `while(` $Cond$ `) do` X'

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain

2: Let O_{aux} be a set of new variable names s.t. there is a one-to-one and onto mapping function $\sigma : O_{aux} \rightarrow Out(X)$

3: Let $t = (N_t, In(X) \cup O_{aux}, Out(X), Pre(X), Eff(X), \emptyset)$ be a task description where N_t is a unique task name

4: Let $\langle t', D' \rangle = Translate\text{-}Construct(X')$

5: Let n' be a task node for task t'

6: Let n_X be a task node for task t // *Used in the recursive step*

7: Let $\lambda_1 = \{\langle n', i \rangle \leftarrow i \mid i \in In(X)\} \cup \{\langle n_X, i \rangle \leftarrow i \mid i \in In(X)\}$
$\cup \{\langle n_X, o_{aux} \rangle \leftarrow \langle n', o \rangle \mid o \in Out(X) \text{ and } o = \sigma(o_{aux})\}$
$\cup \{o \leftarrow \langle n_X, o \rangle \mid o \in Out(X)\}$

8: Let $\lambda_2 = \{o \leftarrow o_{aux} \mid o \in Out(X) \text{ and } o = \sigma(o_{aux})\}$

9: Let $m = (N_m, In(X) \cup O_{aux}, \emptyset, Out(X), Pre(X), Eff(X), \emptyset,$
$[\ Cond : (\{n', n_X\}, \{(n', n_X)\}, \lambda_1);$
$\top : (\emptyset, \emptyset, \lambda_2)])$

10: Add method m to M

11: Add a new concept N_t, a new individual N_m and assertion $N_t(N_m)$ to T_{ont}

12: **return** $\langle t, D \cup D' \rangle$

The complicated part of the translation is to handle the outputs that are generated inside the repeat loop. If the output of the composite process is linked to the output of a process that is performed inside the loop, we need to use the output value generated at the last iteration of the loop. In the translated version, the last recursive call ends with the empty task network. To handle the flow of data correctly, we define additional input parameters in the generated task and methods. These additional input parameters simply carry the output generated at the previous step and in the end the empty task network simply assigns these values

to the output variables. As a result, the final output values are actually the ones coming from the latest iteration.

Repeat-Until The `Repeat-Until` construct repeats a control construct until a certain condition is satisfied. This is the negated version of `Repeat-While` loop with the difference that we will always perform the control constrict at least once because the condition is checked after the perform. Therefore, the translation is very similar to `Repeat-While` but we need an additional method that will perform the first iteration of the loop. After the first iteration, we call the second method that mimics the behavior of `Repeat-While`.

Perform The `Perform` construct is used to execute one other process. The executed process can be atomic or composite. Translating the `Perform` construct can be achieved by simply translating the referred process. Other than that, the perform construct helps us to create the data flow bindings as described earlier.

Translating Processes

Translation of a process generates one HTN-DL task as in the case of control constructs. In addition, an AtomicProcess is mapped to an HTN-DL operator and a CompositeProcess is mapped to an HTN-DL method. Translating an AbstractProcess on the other hand, does not create any operator or method as expected. Since AbstractProcess is not a concrete executable process it is only mapped to an HTN-DL task.

Translation of processes is very straight-forward. The inputs, outputs, preconditions and effects of the process are directly used in the task definition. Similarly, we generate the operator definition for AtomicProcesses and the method definition for CompositeProcesses. The method we generate for a CompositeProcess is simply the method generated for the top-most control construct used in the process description.

4.3 OWL-S semantics

OWL-S specification does not provide a formal semantics for the language. The translation we provided can be used as a basis for a formal semantics of OWL-S. Earlier Narayanan and McIlraith [92] defined a semantics for OWL-S in terms of the situation calculus [106] and Golog [77]. The semantics is given by mapping the process models to actions in the situation calculus formalism. Based on these semantics, a formal definition of Web Service composition problem is given. In this section, we show that the semantics we provided is compatible with this view. More specifically, we show the plans generated by HTN-DL algorithm based on our translation are equivalent to the action sequences found in Situation Calculus.

The situation calculus in a first-order language for reasoning about action and change. In the situation calculus, the state of the world is described by functions and relations (fluents) relativized to a situation s, e.g., $f(x, s)$. The function $do(a, s)$ maps a situation s and an action a into a new situation. A situation is simply a history of the primitive actions performed from an initial, distinguished situation

S_0.

Golog is a high-level logic programming language based on the situation calculus, that enables the representation of complex actions. It builds on top of the situation calculus by providing a set of extra-logical constructs (Figure 4.1) for assembling primitive actions, defined in the situation calculus, into complex actions that collectively comprise a program, δ. Given a domain theory, D and a Golog program δ, program execution must find a sequence $\vec{a}$, such that $D \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that Golog program δ starting at S_0 will legally terminate in situation $do(\vec{a}, S_0))$ where $do(\vec{a}, S_0))$ is used to abbreviate the following expression $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0))$. Thus, $a_1, \ldots, a_n$ are the actions that realize Golog program δ, starting in the initial situation, S_0.

The semantics given in [92] and [88] maps an OWL-S process to a Golog program where atomic processes in OWL-S are mapped to primitive actions in Golog and composite processes in OWL-S are mapped to corresponding complex Golog actions. Note that, there is a representational difference between how HTN-DL and Situation Calculus describe the state of the world. HTN-DL represents the state as an OWL-DL KB whereas in the situation calculus, the state of the world is de-

Syntax	Explanation
a	primitive action
$\delta_1; \delta_2$	sequence
$cond?$	test
$\delta_1 \mid \delta_2$	nondeterministic choice of actions
δ^*	nondeterministic iteration
if *cond* **then** δ_1 **else** δ_2 **endIf**	conditional
while *cond* **do** δ **endWhile**	while loop

Table 4.1: A subset of Golog constructs to create complex actions that are relevant to OWL-S constructs.

scribed by relations (fluents) relativized to a situation which is simply a sequence of actions. As shown in [4], it is straight-forward to translate an OWL-DL KB to an equivalent Situation Calculus theory using the correspondence between DLs and first order logic [13]. Furthermore, [4] shows that, with an appropriate update semantics, after the execution of a service, the respective successor states obtained in Situation Calculus and a DL based action formalism can be proven equivalent. Therefore, we can assume that when the same sequence of actions/operators are applied to a situation/state, the logical entailments of the final situation/state will be the same. Rest of this chapter we will use this equivalence and the same name to denote world states in both notations when the meaning is clear.

Using the Situation Calculus based semantics of OWL-S, the Web Service composition problem is defined as follows:

Definition 4.1 (OWL-S Service Composition) *Let $K = \{K_1, K_2, \ldots, K_m\}$ be a collection of OWL-S processes, C be a possibly composite process defined in K, S_0 be the initial state, and $P = (p_1, p_2, \ldots, p_n)$ be a sequence of atomic processes defined in K. Then P is a composition for C with respect to K in S_0 iff in action theory, we can prove:*

$$\Sigma \models Do(\delta_C, S_0, do(\vec{a}, S_0)))$$

where

- Σ *is the axiomatization of K and S_0 as defined in action theory.*
- δ_C *is the complex action defined for C as defined in action theory*

- *a_i is the primitive action defined for p_i as defined in action theory*

Note that this definition is for offline planning, i.e. there is no execution of information-providing Web Services during planning. In the Golog approach [88], information gathering during planning is achieved by what is called the Middle Ground execution (MG) for sensing actions. The correctness of MG depends on the Invocation and Reasonable Persistence (IRP) assumption [88]. Intuitively, IRP assumption says that

- Information-providing services should be executable in the initial state, and
- Information gathered from these services cannot be changed by external or subsequent actions.

The first condition follows from the fact that information gathering is done with respect to the initial state. The second condition assumes no external source will change the gathered information during the planning process but also prohibits the planner from changing the gathered information as well. This is to prevent the kind of problems we discussed in 3.3.3, that is, the gathered information might incorrectly overwrite the effects of already planned actions. Since [88] is not limited to boolean sensing actions the simple solution we described in 3.3.3 is not applicable any more. If we use the same restrictions for information-providing services in both Golog and HTN-DL then the correspondence between the entailments of HTN-DL state and Situation Calculus situation would be preserved.

We formally state the equivalence between the plans generated in HTN-DL and Situation Calculus as follows:

Theorem 4.1 *Let $K = \{K_1, K_2, \ldots, K_m\}$ be a collection of OWL-S process models, C be a possibly composite process defined in K, S_0 be the initial state, and $P = (p_1, p_2, \ldots, p_n)$ be a sequence of atomic processes defined in K. Then P is a composition for C with respect to K in S_0 iff P is a plan for planning problem (S_0, T_C, D) where T_C is the task network containing the single task returned by the translation for process C, and D is the HTN-DL domain created from K.*

Proof See the Appendix for the proof of this theorem. □

Algorithm 9 *Translate-Repeat-Until*(X)

Inputs: X is in the form: `do` X' `until(` $Cond$ `)`

1: Let $D = (O, M, T_{ont})$ be an initially empty HTN-DL domain

2: Let O_{aux} be a set of new variable names s.t. there is a one-to-one and onto mapping function $\sigma : O_{aux} \rightarrow Out(X)$

3: Let $t_1 = (N_{t_1}, In(X), Out(X), Pre(X), Eff(X), \emptyset)$ be a task description where N_{t_1} is a unique task name

4: Let n_1 be a task node for task t_1

5: Let $t_2 = (N_{t_2}, In(X) \cup O_{aux}, Out(X), Pre(X), Eff(X), \emptyset)$ be a task description where N_{t_2} is a unique task name

6: Let n_2 be a task node for task t_2

7: Let $\langle t', D' \rangle = Translate\text{-}Construct(X')$

8: Let n' be a task node for task t'

9: Let $\lambda_1 = \{\langle n', i \rangle \leftarrow i \mid i \in In(X)\} \cup \{\langle n_2, i \rangle \leftarrow i \mid i \in In(X)\}$
$\cup \{\langle n_2, o_{aux} \rangle \leftarrow \langle n', o \rangle \mid o \in Out(X) \text{ and } o = \sigma(o_{aux})\}$
$\cup \{o \leftarrow \langle n_2, o \rangle \mid o \in Out(X)\}$

10: Let $\lambda_2 = \{o \leftarrow o_{aux} \mid o \in Out(X) \text{ and } o = \sigma(o_{aux})\}$

11: Let $m_1 = (N_{m_1}, In(X), \emptyset, Out(X), Pre(X), Eff(X), \emptyset,$
$[\top : (\{n', n_2\}, \{(n', n_2)\}, \lambda_1)])$

12: Let $m_2 = (N_{m_2}, In(X) \cup O_{aux}, \emptyset, Out(X), Pre(X), Eff(X), \emptyset,$
$[\ Cond : (\emptyset, \emptyset, \lambda_2);$
$\top : (\{n', n_2\}, \{(n', n_2)\}, \lambda_1)])$

13: Add methods m_1 and m_2 to M

14: Add new concepts N_{t_1} and N_{t_2}, new individuals N_{m_1} and N_{m_2}, and assertions $N_{t_1}(N_{m_1})$ and $N_{t_2}(N_{m_2})$ to T_{ont}

15: **return** $\langle t, D \cup D' \rangle$

Chapter 5

Optimizing OWL-DL Reasoning

So far, the thesis focused on the coupling of DLs with HTNs for solving Web Service composition problems. Proposed HTN-DL formalism uses DL reasoning services for task matching and precondition evaluation. A composition system built on HTN-DL formalism can be used in practice only if the DL reasoning can be done efficiently.

In particular, this chapter focuses on efficient reasoning with nominals, a topic that has not been investigated in the DL literature. At the current stage of research and deployment, existing optimizations have been implemented and proved useful for the DL $\mathcal{SHIN}$. However, there was no specific optimization techniques developed to handle nominals. Furthermore, even though a decision procedure for $\mathcal{SHON}$ fragment of OWL-DL, which includes nominals but not inverse properties, was known since 2001, there was no implemented system that handled this expressivity (with or without optimizations).

Nominals allow us to define concepts in terms of individuals and are required for describing many different aspects of Web Services. For example, a language translator Web Service such as the one provided by Babelfish does not only say that the input should be an instance of the `Language` concept but specifies the exact set of supported languages as an enumeration, e.g. {`English,French,...,Russian`}.

Another example is when a service is only executable by people located at a certain geographic location;. For example, OWL-S profile ontology contain a service parameter named `geographicRadius`[1] to limit this region to a specific country. Then the set of services executable in the US would be defined as the concept $\exists geographicRadius.\{\texttt{US}\}$ where an enumeration with a single element is used.

One other use of nominals in HTN-DL is to partially "close the world". A known disadvantage of open world reasoning is that it is not possible to say that we have complete knowledge about some aspect of the domain. Combined with the information gathering ability, open world reasoning results in what is known as *sensor abuse* [83, 40]; that is, the planner tries to gather more and more information which does not help in finding a new plan. For example, consider the problem of buying an airplane ticket from Washington, DC to Kyoto, Japan. We know that there are only three airports located in the Washington, DC area that are identified by the airport codes {DCA,IAD,BWI}. Under open world reasoning, the planner would think that there might be other airports in the area that we are not aware. In the case that no tickets can be found for the flights departing from these airports, the planner would execute information-providing services to find information about additional airports. Obviously, gathering information about other airports (that are located in different parts of the country) will not help us to find a plan. Using nominals, we can state the known three airports are indeed the only airports in the area preventing such problems.

[1]This parameter was previously part of the core OWL-S ontologies but is now provided in one of the extension ontologies developed by the OWL-S coalition

In this chapter, we will first describe why reasoning with nominals is challenging and explain how some of the existing optimization techniques are not applicable any more. Then we will describe several different optimization techniques that are designed to improve the efficiency of standard reasoning services such as KB consistency, classification, and realization. In Chapter 6 we will examine the case of answering conjunctive queries.

5.1 Reasoning with Nominals in OWL-DL

OWL-DL can be seen as a syntactic variant of the DL $\mathcal{SHOIN}$. Although tableau-based decision procedures for prominent fragments of $\mathcal{SHOIN}$, such as $\mathcal{SHIN}$ [60] and $\mathcal{SHON}$ [56] have been known for quite a long time, the design of a decision procedure for $\mathcal{SHOIN}$ has been accomplished only very recently [62].

Expressive description logics, in particular the ones mentioned above, are known to have very high worst-case complexity. As a consequence, there exists a significant gap between the design of a decision procedure and the achievement of a practical implementation. Naive implementations are doomed to failure. In order to achieve acceptable performance, modern DL reasoners, such as FaCT++, RACER, DLP and Pellet, implement a suite of *optimization techniques* [54, 61, 53, 47, 48, 46]. These optimizations lead to a significant improvement in the empirical performance of the reasoner and have proved effective in wide variety of realistic applications.

From an implementation point of view, reasoning with OWL-DL is hard because the existence of nominals in the language pose some serious challenges. For

example, in the presence of nominals, ABox assertions can affect the satisfiability of a concept and the classification of a TBox. In other words, nominals break the "separation" between the TBox and the ABox that traditionally existed in the implemented DLs. For example, one optimization technique is partitioning the ABox into smaller disconnected components where consistency checking is faster. A query about an individual can be answered by the partition it belongs to. However, when there are nominals, it is not possible to find the partitions in a preprocessing step since nominals may connect different parts of the ABox. Another optimization technique that has been reported to be very effective is *ABox chain contraction* [46]. This technique removes the individuals from an ABox by turning them into $\exists$ restrictions on the predecessor individuals. Consequently, the size of the ABox is reduced and the effect of model caching optimization is increased. However, such a contraction is not possible for individuals used in concept descriptions.

From a logical point of view, the *nominal constructor* [56, 109] transforms the object name o into the concept description $\{o\}$, which is evaluated, by every model-theoretic interpretation, to a singleton set with o as its only element. So far, nominals have been partially approximated in DL reasoners by treating them as pair-wise disjoint atomic concepts, commonly called *pseudo-nominals*. However, this technique is known to lead to incorrect inferences in some cases.

From a modeling point of view, nominals are used in a significant number of ontologies available on the Semantic Web. The OWL-DL specification [23] contains two modeling constructs specific for nominals, which illustrate their main uses in Ontology Engineering.

- The *OneOf* construct allows to define a concept by finite enumeration of its elements. For example, the atomic concept *Continent* can be defined, using nominals, as follows:

$$Continent \equiv \{europe, asia, america, antartica, africa, oceania\}$$

  where the elements of the enumeration are individuals in the KB.

- The *hasValue* construct is used as a shorthand for an existential restriction on a nominal concept. This construct can be used to describe European cities as cities located in Europe:

$$EuropeanCity \sqsubseteq City \sqcap \exists locatedIn.\{europe\}$$

One prominent example of the use of nominals for modeling is the Wine Ontology [112], the ontology prepared by the W3C Web Ontology (WebOnt) working group and published in the OWL guide [112] to demonstrate the features of the language. The purpose of the WebOnt group was to use all the language constructs of OWL in a relatively straight-forward way to teach novice OWL users about OWL. However, the resulting ontology turned out to be very challenging for automated reasoners. This feature, simple in design but very hard to reason with, makes the Wine ontology a very interesting test case. For this reason, we will examine the Wine ontology in more detail and explain the novel optimizations developed in this thesis using examples from the Wine ontology. Although the Wine ontology might not seem directly related to HTN-DL it is believed to be a good representative of

OWL ontologies (as witnessed by the fact that the developers of the OWL language created it as a teaching tool) and other OWL ontologies (that are more relevant for HTN-DL) will have similar characteristics.

The Wine ontology extensively relies on the *OneOf* and *hasValue* constructs for describing different kinds of wines according to various criteria, like the area they are produced in, the kinds of grapes they contain, their flavor and color, etc. For example, a "Cabernet Franc Wine" is defined to be a dry, red wine, with moderate flavor and medium body and which is made with Cabernet Franc grapes

$$
\begin{aligned}
CabernetFranc &\equiv Wine \sqcap \leq 1 madeFrom \sqcap \exists madeFrom.\{cabFrancGrape\} \\
CabernetFranc &\sqsubseteq \exists hasColor.\{red\} \sqcap \exists hasFlavor.\{moderate\} \sqcap \\
&\quad \exists hasBody.\{medium\}
\end{aligned}
$$

Potential wine flavors, colors, etc are defined using an enumeration of individuals. For example:

$$WineFlavor \equiv \{delicate, moderate, strong\}$$

The Wine ontology contains only 138 concepts and 206 individuals and hence it is a relatively small knowledge base. However, its classification has remained, so far, an open problem for DL reasoners. Using the optimizations proposed here, the reasoner Pellet has become the first (and currently the only) reasoner to classify the Wine ontology.

There are several reasons that make the Wine ontology hard for automated reasoning: First, as we mentioned earlier, there is the issue of ABox statements affecting the TBox. Second, the ontology contains a significant number of General Concept Inclusion Axioms (GCIs) associated with nominals that cannot be handled by current preprocessing techniques. As a result, tableau expansions become very expensive computationally and hence every additional satisfiability test performed during classification is likely to be very expensive.

5.2 Preprocessing Optimizations

The axioms in a DL KB are not typically in a form that facilitates reasoning services. For example, semantically equivalent but syntactically different axioms are hard to detect by a reasoner and they degrade the performance of reasoning. Therefore, modifying the axioms by applying syntactic transformations in a preprocessing step has proved to be very useful in practice [54]. In the following subsections, we will first review the existing preprocessing optimization *absorption* and then describe a novel technique *nominal absorption* that can absorb axioms involving nominals.

5.2.1 Existing Optimizations

General Concept Inclusion Axioms (GCIs) are hard to reason with, given the high degree of non-determinism they introduce. For each GCI, one disjunction is added to the label of *each* node in a tableaux expansion, which causes an exponential blow-up in the search space. As a consequence, even a reduced number of GCIs can

degrade the performance of a DL reasoner significantly.

Primitive definitions, on the other hand, can be efficiently handled by the technique known as *lazy unfolding* [54]. Instead of internalizing primitive definitions and causing additional disjunctions one can have additional expansion rules that will replace defined concepts with their definition. For example, if $\mathcal{T}$ contains the non-primitive definition axiom $A \equiv C$, and the $\sqcap$-rule is applied to a concept $(A \sqcap D) \in \mathcal{L}(x)$ so that A and D are added to $\mathcal{L}(x)$, then at this point A can be unfolded by substituting it with C.

Although it is possible to treat primitive definitions as general axioms this is highly inefficient. The solution to handle TBoxes that contain both primitive definitions and general axioms is to divide the TBox into two components, an unfoldable part $\mathcal{T}_u$ and a general part $\mathcal{T}_g$, such that $T_g = \mathcal{T} \setminus \mathcal{T}_u$, and $\mathcal{T}_u$ contains unique, acyclical, definition axioms. It is then possible to use lazy unfolding to deal with $\mathcal{T}_u$, and internalization to deal with $\mathcal{T}_g$.

Absorption [54] is a preprocessing technique that tries to eliminate GCIs from a TBox by replacing them with primitive definitions. This technique moves axioms from $\mathcal{T}_g$ into $\mathcal{T}_u$ reducing the number of disjunctions that will be introduced during tableaux expansion. Let us illustrate how this technique works with the following general axiom

$$MealCourse \sqcap \exists hasFood.Dessert \sqsubseteq \forall hasDrink.(\exists hasSugar.\{Sweet\})$$

that says every meal course containing a dessert should have a sweet wine. We can

transform this axiom into a primitive definition of the form

$$MealCourse \sqsubseteq \forall hasDrink.(\exists hasSugar.\{Sweet\}) \sqcup \neg\exists hasFood.Dessert$$

that says every meal course has either sweet wine or it cannot contain a desert. The former axiom introduces a disjunction for every node in the completion graph whereas the disjunction in the the latter axiom is only applied to nodes that has *MealCourse* in its label. This way the non-determinism is localized to a much smaller subset.

Absorption has revealed a key technique in the past for processing DL ontologies [53, 55]. However, existence of nominals in the KB causes a different kind of GCIs to occur which are not amenable with the existing absorption technique. In the following section, we describe novel transformation methods to overcome this problem.

5.2.2 Nominal Absorption

As stated before, there are two main ways of using nominals in ontologies: defining concepts by finite enumeration of its elements (the OWL *OneOf* construct) and defining concepts in terms of existential restrictions on a nominal (the OWL *hasValue* construct). For both cases, we provide an extension of existing absorption techniques.

OneOf Absorption Let us start with enumerations. Consider the concept named *WineColor* in the Wine Ontology, which is defined as follows:

$$WineColor \equiv \{red, rose, white\}$$

$$WineColor \sqsubseteq WineDescriptor$$

The combination of these two axioms create a GCI ($\{red, rose, white\} \sqsubseteq WineDescriptor$) which is not captured by the currently available absorption techniques and hence, the disjunction:

$$\neg WineColor \sqcup \{red, rose, white\}$$

would be added to every node in the tableau expansion. On the other hand, an enumeration is equivalent to the disjunction of its elements, i.e.:

$$\{rose, red, white\} \equiv \{rose\} \sqcup \{red\} \sqcup \{white\}$$

This leads to an additional difficulty: enumerations are likely to introduce a significant number of backtracking points. These disjunctions, when added to every node of the tableau expansion, cause the search space to grow exponentially with the number of elements in the enumeration. Thus, the presence of these non-absorbable GCIs is going to significantly affect reasoning performance.

Nominal absorption is a novel optimization technique that transforms such

axioms into a primitive definition and a set of ABox assertions. The technique relies on the following equivalence:

Proposition 5.1 *The inclusion axiom (5.1) is logically equivalent to the set of TBox axioms and ABox assertions in (5.2)*

$$C \equiv \{a_1, \ldots, a_n\} \tag{5.1}$$

$$C \sqsubseteq \{a_1, \ldots, a_n\} \textit{ and } C(a_1) \textit{ and } \ldots \textit{ and } C(a_n) \tag{5.2}$$

Proof See the Appendix for the proof of this proposition. □

This proposition lets us to replace a non-absorbable GCI into one primitive definition and a set of ABox assertions. Note that the set $C(a_1), ..., C(a_n)$ of ABox assertions is equivalent to the GCI $\{a_1, ..., a_n\} \sqsubseteq C$. In our example, the enumeration axiom would be absorbed as follows:

$$WineColor \sqsubseteq \{red, rose, white\}$$

$$WineColor(red); WineColor(rose); WineColor(white)$$

We still have a disjunction due to the presence of $\{red, rose, white\}$. However, this disjunction will only affect the instances of *WineColor* concept instead of all the individuals. Thus, the effect of the disjunction is now localized to a very small number of individuals.

HasValue Absorption Let us now consider the case of *hasValue* restrictions. Axioms in the following form are commonly found in the Wine ontology:

$$Riesling \equiv Wine \sqcap \leq 1 madeFrom \ \sqcap \exists madeFrom.\{RieslingGrape\}$$

Considering that there are other inclusion axioms such as

$$Riesling \sqsubseteq \exists hasColor.\{White\}$$

we are again left with GCIs. Standard absorption techniques can take care of such cases by absorbing the axiom into the definition of the *Wine* concept, i.e. the concept

$$Wine \sqsubseteq Riesling \sqcup \forall madeFrom.\neg\{RieslingGrape\} \sqcup \geq 2 madeFrom)$$

is added to the definition of *Wine*. Therefore, this disjunctive definition introduces a backtracking point in the tableau expansion for every node containing *Wine* in its label. Standard absorption technique creates nearly 30 of such disjunctions relative to the *Wine* concept, and since there are more than 50 *wine* instances in the ontology, the search space significantly grows.

However, the semantics of nominals allows a more effective absorption of the above axiom by taking advantage of the following equivalence:

Proposition 5.2 *The following two inclusion axioms are logically equivalent:*

$$\exists p.\{o\} \sqsubseteq C \tag{5.3}$$

$$\{o\} \sqsubseteq \forall p^{-}.C \tag{5.4}$$

Proof See the Appendix for the proof of this proposition. □

It is very straight-forward to show that the inclusion axiom $\{o\} \sqsubseteq C$ is logically equivalent to the ABox assertion $C(o)$ (see the proof of Proposition 5.1 in the appendix). Using these equivalences in the previous example would yield the following ABox assertion:

$$(\forall madeFrom^{-}.(Riesling \sqcup \neg Wine \sqcup \geq 2 madeFrom))(RieslingGrape)$$

The resulting axiom still contains the same number of disjuncts, but this time the effect is localized to the individuals related to *RieslingGrape* via the role *madeFrom*, which are considerably less than the number of *Wine* instances.

Algorithm 10 describes the standard absorption algorithm extended with nominal absorption. The additional steps are marked with comments in bold font.

5.3 Optimizations for Consistency Checking

After the preprocessing step, a KB consistency check is done by applying the tableau expansion rules of Table 2.2 to the initial completion graph. As explained in Section 2.3.4, disjunctive concepts give rise to non-deterministic expansion. Searching non-deterministic expansions is the main cause of intractability in tableaux

Algorithm 10 Absorb($C \sqsubseteq D$)

1: Let $G = \{C, \neg D\}$ be the initial set // *Initialize*
2: **if** $A \in G$ and A is atomic **then** // *Concept absorption*
3: Replace $(A \sqsubseteq C) \in T_u$ with $A \sqsubseteq \sqcap\{C, \neg(\sqcap(G \setminus \{A\}))\}$
4: **return**
5: **end if**
6: **if** $C \in G$ and $C = \{o_1, \ldots, o_n\}$ **then** // ***OneOf absorption***
7: Add $(\neg(\sqcap G))(o_i)$ to the ABox for each individual o_i
8: **return**
9: **end if**
10: **if** $C \in G$ and $C = \exists p.\{o_1, \ldots, o_n\}$ **then** // ***HasValue absorption***
11: Add $(\forall p^{-}.\neg(\sqcap G))(o_i)$ to the ABox for each individual o_i
12: **return**
13: **end if**
14: **if** $A \in G$ (resp. $\neg A \in G$) and $(A \equiv D) \in T_u$ **then** // *Simplification*
15: Substitute A (resp. $\neg A$) with D (resp. $\neg D$) and go to line 2
16: **end if**
17: **if** $C \in G$ and $C = (C_1 \sqcap \ldots \sqcap C_n)$ **then** // *Conjunction simplification*
18: Let $G = G \sqcup \{C_1, \ldots, C_n\}$ and go to line 2
19: **end if**
20: **if** $C \in G$ and $C = (C_1 \sqcup \ldots \sqcup C_n)$ **then** // *Recursion*
21: **for all** C_i **do**
22: Try recursively absorbing $\neg C_i \cup G \setminus \{C\}$
23: **end for**
24: **else**
25: Leave G in T_g (Absorption failed)
26: **end if**

subsumption testing algorithms. Next, we will describe some of the existing optimizations developed to tackle this issue and the problems with these optimizations.

5.3.1 Existing Optimizations

Searching non-deterministic expansions due to disjunctive concepts can be very expensive especially when there is an inherent unsatisfiability concealed in a sub-problem. This can lead to large amounts of unproductive backtracking search, sometimes called thrashing. For example, expanding a node x, where

$$\mathcal{L}(x) = \{(C_1 \sqcup D_1), \ldots, (C_n \cup D_n), \exists R.(A \sqcap B), \forall R.\neg A\}$$

could lead to the fruitless exploration of 2n possible R-successors of x before the inherent unsatisfiability is discovered.

Backjumping [54], a type of *dependency-directed backtracking*, is an optimization technique that associates a *dependency set* with the node and edge labels in a completion graph to indicate the non-deterministic choice on which the labels depend. When a clash is discovered, the dependency sets of the clashing concepts can be used to identify the most recent branching point causing the clash. It is then possible to jump back over intervening branching points *without* exploring any alternative branches. This technique can lead to a dramatic reduction in the size of the search tree and thus a huge performance improvement.

When a disjunction in the label of the node is being expanded, the order in which disjuncts are selected can make a dramatic change in the performance

of the tableau reasoner. Many different heuristics have been developed for DPLL SAT algorithms to minimize the size of the search tree. The well known Maximum number of Occurrences in disjunctions of Minimum Size (MOMS) heuristic [33] and the heuristic from Jeroslow and Wang [69] are two examples. However, it has been shown in the DL literature that such heuristics generally counter-interact with backjumping leading to much worse performance [54].

In the next subsection, we will present *Learning-based Disjunct Selection*, a novel disjunct selection heuristic that does not have an adverse effect on other optimizations. Unlike other optimization techniques, learning-based disjunct selection is aimed towards building clash-free completion graphs rather than revealing clashes. Nearly all of the ontologies published on the Semantic Web are consistent (and inconsistent ontologies are quite useless as they entail every logical sentence). For this reason, KB consistency checks end up building clash-free completion graphs and learning-based disjunct selection speeds up this process.

5.3.2 Learning-based Disjunct Selection

An investigation of real world ontologies reveals that, in many cases, there are some disjunctions that essentially have only one possible expansion. However, this reality is detected by the reasoner only after numerous tableaux rule applications that tries all the "wrong" disjuncts. Moreover, this expensive cycle is typically repeated many times for each individual with similar characteristics. Let us illustrate this case with an example from OWL-S ontologies. Given the following axioms

$$Process \equiv AtomicProcess \sqcup CompositeProcess \sqcup SimpleProcess$$

$$AtomicProcess \sqsubseteq \neg SimpleProcess$$

$$AtomicProcess \sqsubseteq \neg CompositeProcess$$

$$CompositeProcess \sqsubseteq \neg SimpleProcess$$

$$CompositeProcess \equiv Process \sqcap \leq 1.composedOf \sqcap \geq 1.composedOf$$

$$\top \sqsubseteq \forall composedOf.ControlConstruct$$

$$\top \sqsubseteq \forall composedOf^{-}.CompositeProcess$$

The standard preprocessing steps, e.g. normalization and absorption, produce the following axiom[2]:

$$Process \sqsubseteq \geq 2.composedOf \sqcup CompositeProcess \sqcup \leq 0.composedOf$$

During the tableaux expansion, for any *AtomicProcess* instance, we will face to expand this disjunction. Obviously, in this example, the only right selection is the second disjunct ($\leq 0.composedOf$) because the first disjunct ($\geq 2.composedOf$) is unsatisfiable by definition (due to the domain restriction only *CompositeProcess*es can have *composedOf* roles and *CompositeProcess*es are not allowed to have more than one value) and the second disjunct (*CompositeProcess*) causes a clash due to the disjointness axiom between *AtomicProcess* and *CompositeProcess*. However, a DL reasoner will observe this fact only after applying several other rules, in this case the $\geq$*-rule* and *unfolding-rule*. When these rule applications are interleaved

[2]There is a possibility that absorption algorithm yields different results depending on the order axioms processed, but the non-determinism does not have any effect on this specific example

with other rule applications, several other disjunctions might have been expanded for a different number of individuals, which causes a significant amount of wasted computation. Moreover, OWL-S knowledge bases would typically have lots of *AtomicProcess* instances and, consequently, these steps would be repeated for each of such instances, which degrades performance significantly.

The learning-based disjunct selection technique aims to minimize the wasted computation by avoiding inherently clash-generating expansions. The idea is to reuse the clash-free expansions for instances with similar characteristics. The heuristic is to sort the disjuncts based on how many clashes they caused during rule applications. Note that when the dependency sets for concepts are being maintained it is quite easy to detect if a certain disjunction expansion caused the clash or not.

Algorithm 11 shows the pseudo-code of learning based disjunct selection. This technique only learns from clashes, i.e. unsuccessful selections, and it does not keep track of successful expansions. It would be nearly impossible to keep track of successful expansions during completion since it is not clear when and how we can conclude a disjunction expansion was successful. One possibility is to do a post-processing step after a clash-free completion and iterate through the nodes in the completion graph to update the disjunction statistics for future use.

5.3.3 Completion Graph Caching

In the presence of nominals in the TBox, ABox assertions can affect concept satisfiability and classification. Thus, when checking the satisfiability of an atomic

Algorithm 11 expand-disjunction(x, D)

Inputs: x the node we are expanding in the completion graph, D is a disjunctive concept in the form $D_1 \sqcup \ldots \sqcup D_n$

1: Let $stats$ = get-statistics(D) *// stats is an array of integer values*
2: **if** stats not found **then**
3: Let $stats$ be an integer array of length n
4: **for all** $i : 1 \leq i \leq n$ **do**
5: Let $stats[i] = 0$
6: **end for**
7: save-statistics(D, $stats$)
8: **end if**
9: Pick the next untried disjunct D_k such that stats[k] is minimum
10: Add D_k to $\mathcal{L}(x)$ and continue tableau expansion
11: **if** there is a clash **then**
12: increment $stats[k]$
13: **end if**

concept A after the initial KB consistency check, we need, in principle, to include in the initial completion graph for A a root nominal node x_a for each individual a in the ABox. The presence of these nodes in the initial configuration of the graph is likely to cause a large number of expansion rules to be triggered and hence may involve a significant computational overhead.

The main idea underlying the completion graph caching technique is to store the state of the completion graph *after* the initial KB consistency check and reuse it for subsequent concept satisfiability and subsumption tests. Expanding the nominal nodes from its initialization state may involve the application of a large number of expansion rules. By using cached graph we avoid repeating the process for different concept satisfiability tests.

For the initial KB consistency test, we create all the nominal nodes and apply

all the expansion rules. For any subsequent consistency check, we use the already expanded graph as the initial graph so that already applied expansion rules will not be repeated.

One needs to be careful when reusing an earlier completion graph because there might be some edges or node labels dependent on a non-deterministic choice. If there is a clash due to such an edge or a node label, the backtracking must be done accordingly. In order to backtrack correctly, we need to cache not only the nodes and edges, but also the information about dependency sets for the labels of nodes and edges plus the history of merge operations so that nodes can be restored after backjumping. Although caching this information affects memory consumption, the overhead is not critical and pays off in terms of significant speed-up in subsequent concept satisfiability and subsumption tests.

5.3.4 Lazy Completion Graph Generation

Even in the presence of nominals in the TBox, there are typically many atomic concepts whose corresponding satisfiability check *does not* involve the application of the nominal rule and, therefore, the content of the ABox and the nominals do not influence their satisfiability. For these concepts, generating the nominal nodes corresponding to the ABox individuals results in an unnecessary overhead. Even if we use the cached completion graph for these individuals, maintaining extra nodes (copying, checking if a rule is applicable, etc.) can be costly.

Since the KB is consistent, the rules triggered by the presence of the initial

graph of nominal nodes will never yield to a clash in the tableau expansion for A.

Lazy completion graph generation avoids such a computational burden by *not* including the nominal nodes in the initial completion graph when checking concept satisfiability. If the nominal rule is triggered during tableau expansion, then all the nominal nodes are added to the completion graph. This simple technique may yield a dramatic performance improvement, as discussed later on in empirical results section.

It is important to realize that the combination of lazy completion graph generation and completion graph caching may interact with backjumping and, in order to ensure the correctness of the technique, we generate the initial set of nominal nodes every time backjumping is applied, even if the nominal rule has not been triggered.

The reader may have noted that lazy completion graph generation is very conservative in two different ways: First, even if a merge is forced by the application of the nominal rule, there are cases in which it suffices to generate only a *subset* of the nominal nodes; second, the generation of the completion graph may not always be required after backjumping. This provides room for further improvements in the future.

5.4 Optimizations for Subsumption and Instance Checking

Classification of named concepts in a KB is one of the most important applications of DL reasoners. Optimization techniques for classification aim at reducing as much as possible the number of subsumption tests to be performed. Similarly,

for instance checking, we would like to conclude if an individual is an instance of the concept or not without doing a consistency test.

In the next sections, we will first describe how model merging technique can be improved to take advantage of *HasValue* restrictions. Then, we will show this technique can also be used for ordinary existential restrictions. Finally, we will describe a simple method to use these techniques more effectively for defined concepts.

5.4.1 Nominal-based Model Merging

Nominal-based pseudo-model merging is an optimization technique to discover "obvious" non-subsumptions between concepts (or non-instantiations between individuals and concepts). In particular, this technique is especially effective if there are many concepts in the KB defined in terms of *hasValue* restrictions, i.e. existential restrictions on nominals. For example, the concept:

$$RedWine \sqsubseteq Wine \sqcap \exists hasColor.\{red\}$$

is defined in terms of the nominal concept $\{red\}$.

The nominal-based pseudo-model merging technique uses cached information relative to nominals from previous satisfiability tests to prove non-subsumption without performing a new satisfiability test.

The basic idea is to examine the edges from the blockable root node to nominal nodes in the completed graph which was generated to check the satisfiability of a concept. For example, checking the satisfiability of concept $RedWine$ starts by

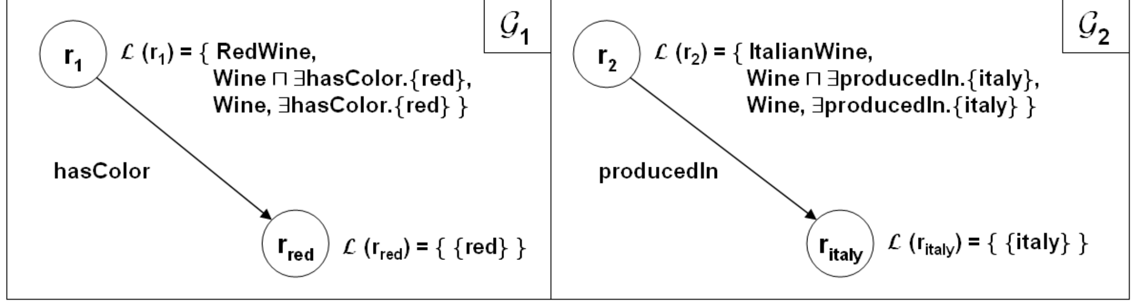

Figure 5.1: Completion graphs for concepts *RedWine* and *ItalianWine*

creating a completion graph that contains a root node r_1 labeled with concept $RedWine$ and one nominal node for each nominal occurring in the ontology. The completion graph $\mathbf{G}_1$ for concept *RedWine* is schematically shown in Figure 5.1. The root node r_1 in $\mathbf{G}_1$ is connected to the nominal node r_{red} through a *hasColor*-labeled edge indicating that $RedWine \sqsubseteq \exists hasColor.\{red\}$. Now let us consider Italian wines, defined as follows:

$$ItalianWine \sqsubseteq Wine \sqcap \exists producedIn.\{italy\}$$

In the completion graph of *ItalianWine* (shown as $\mathbf{G}_2$ in Figure 5.1), the nominal node r_{red} is not a neighbor of the concept node r_2. From this information, it is possible to infer that $\mathcal{O} \not\models ItalianWine \sqsubseteq \exists hasColor.\{red\}$ and thus $\mathcal{O} \not\models ItalianWine \sqsubseteq RedWine$. Note that, for non-simple roles, instead of testing for node neighborhood, we would have considered paths connecting the root node and the nominal node.

However, there is still one more issue we need to consider. Let us consider the

following axioms:

$$DryWine \equiv Wine \sqcap \exists hasSugar.\{dry\}$$

$$NonSweetWine \equiv Wine \sqcap \exists hasSugar.\{dry, offdry\}$$

We want to test whether *DryWine* is subsumed by *NonSweetWine*. The graphs $\mathbf{G}_1$ and $\mathbf{G}_2$ in Figure 5.2 are valid completion graphs for *DryWine* and *NonSweetWine* respectively. The root node r_1 for the concept *DryWine* in $\mathbf{G}_1$ is connected to the nominal node r_{dry} by a *hasSugar*-edge. On the other hand, in $\mathbf{G}_2$, the nominal node r_{dry} is not neighbor of the root node r_2. A naive application of nominal-based pseudo model merging would incorrectly conclude that $DryWine$ is not a subclass of $NonSweetWine$.

In this case, the subsumption holds although the edges to nominal nodes differ. The reason is that there is another valid completion graph ($\mathbf{G}_3$ in Figure 5.2) for $NonSweetWine$ in which the root node r_3 for concept $NonSweetWine$ *does* have a *hasSugar*-edge leading to the nominal node r_{dry}. Therefore, in order to infer the non-subsumption, the edge to the nominal node should be present in *every* possible completion graph for $NonSweetWine$ or, in other words, the presence of the edge should not depend on a non-deterministic choice in the execution of the tableau algorithm. For this reason, nominal-based pseudo-model merging can be used only in conjunction when dependency sets are stored for each node label and edge label. Since all the existing DL reasoners already make use of the dependency-directed backjumping optimization, this requirement does not cause an extra overhead.

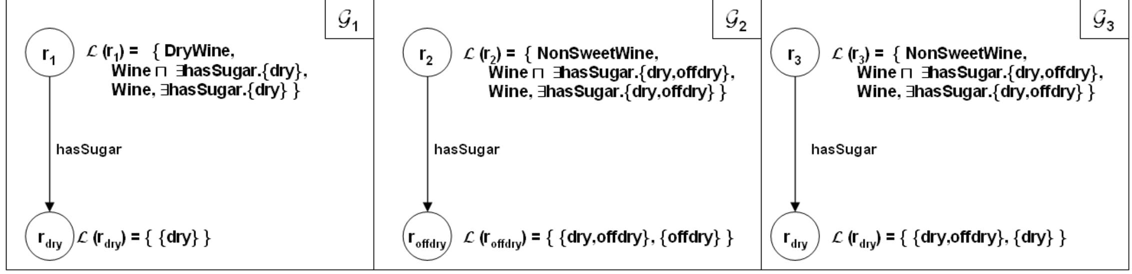

Figure 5.2: Completion graphs for concepts *DryWine* and *NonSweetWine*

Let us now describe formally how the nominal-based pseudo-model merging technique works: Let $\mathbf{G} = (V, E, \mathcal{L}, \neq)$ be a clash-free completion graph for concept A w.r.t. to an ontology $\mathcal{O}$ and $r_A \in V$ be the root node create for concept A[3] that was initialized with $\mathcal{L}(r_A) = \{A\}$. For each nominal o in $\mathcal{O}$ we are guaranteed to have a nominal node $r_o \in V$ such that $\{o\} \in \mathcal{L}(r_o)$.

Suppose that we want to test whether an ontology $\mathcal{O}$ entails the subsumption relation $D \sqsubseteq C$. Let $\mathbf{G}_C$ (respectively $\mathbf{G}_D$) be a fully expanded and clash-free tableaux expansion representing a common model of C and $\mathcal{O}$ (respectively a common model of D and $\mathcal{O}$). Then we say that $\mathcal{O} \not\models D \sqsubseteq C$ if one of the following two conditions hold:

1. There is a *simple* role p such that:

   (a) The nominal node r_o is a p-neighbor of the root node r_C in $\mathbf{G}_C$ and the presence of such an edge does not depend on a non-deterministic choice, and

[3] Note that, there is a possibility that the root node r_A will not exist in the final completion graph $\mathbf{G}_A$ because it was merged into a nominal node and then pruned from the graph. If r_A was merged to another node r_o and the merge operation did not depend on any non-deterministic choice we can simply use r_o instead. But if the merge operation depends on a non-deterministic choice then we cannot use the nominal-based model merging technique.

(b) The nominal node r_o is not a p-neighbor of r_D in $\mathbf{G}_D$.

2. There is a non-simple role p such that:

   (a) There is a path of nodes $z_0, \ldots, z_k$ in $\mathbf{G}_C$ with $k \geq 1, r_C = z_0, r_o = z_k$ and z_i a q-neighbor of z_{i-1} for $0 \leq i < k$ for some q a sub-role of p. Moreover, the presence of such a path does not depend on a non-deterministic choice, and

   (b) There is no such path in $\mathbf{G}_D$ (with or without dependencies) from r_D to the nominal node r_o.

Intuitively, conditions (1a) and (2a) imply that the concept C is subsumed by $\exists p.\{o\}$ and conditions (1b) and (2b) imply that that concept D is not subsumed by $\exists p.\{o\}$. The following theorems (proved in the appendix) state the correctness of this technique.

Theorem 5.1 *Let $\mathbf{G}' = (V', E', \mathcal{L}', \neq)$ be the initial completion graph for the concept C w.r.t the ontology $\mathcal{O}$ such that $V' = \{r_C, r_{o_1}, \ldots, r_{o_m}\}$ where r_C is the root node for concept C and r_{o_i} is the nominal node corresponding to nominal o_i. $\mathcal{L}'$ is initialized such that $\mathcal{L}(r_C) = \{C\}$ and $\mathcal{L}(r_{o_i}) = \{o_i\}$ for $1 \leq i \leq m$.*

Theorem 5.2 *Let $\mathcal{O} \models C \sqsubseteq \exists p.\{o\}$ with C satisfiable w.r.t. $\mathcal{O}$, then in every clash-free and complete graph $\mathbf{G}$ for C w.r.t. $\mathcal{O}$ there must exist a blockable node x with no predecessors (i.e. a root) that verifies the following:*

- *If p is simple then the nominal node o must be a p-neighbor of x in* $\mathbf{G}$

- *If* p *is not simple, then there must exist a path* $z_0, \ldots, z_k$ *in* $\mathbf{G}$ *with* $k \geq 1, x = z_0, o = z_k$ *and* z_i *a* q*-neighbor of* z_{i-1} *for* $0 \leq i < k$ *and* $q \sqsubseteq^*_{\mathcal{R}} p$*.*

Proof See the Appendix for the proof of this theorem. □

One direct application of the nominal-based model merging technique is for getting the role fillers of an individual. For example, the fillers of role *madeIntoWine* for the individual *MerlotGrape* would give us all the wine instances that were made from *MerlotGrape*. Answering this question is equivalent to retrieving the instances of the concept $\exists madeIntoWine^{-}.\{MerlotGrape\}$ which means we can directly use nominal-based model merging.

Chapter 6

Efficient Conjunctive Query Answering

Conjunctive query answering service is an important part of the HTN-DL algorithm. There are two different uses of query answering in HTN-DL. First is the evaluation of preconditions in operator and method descriptions (see Section 3.3.1). The applicability of an action is determined by checking the precondition expression against the current state of the world. The precondition is in the form of a DL conjunctive query and the state is a DL KB; therefore, condition evaluation is simply reduced to query answering. Precondition evaluation either requires answering boolean queries (as in operator preconditions) or retrieval queries (as in method preconditions). Second use of query answering is in task matching (see Section 3.2.3). We explained how task matching relies on determining query subsumption between precondition and effect expressions. Since query subsumption service is reduced to query answering (see Section 2.3.3) the planner is essentially faced with another query answering problem.

During a plan generation, the planner typically evaluates many preconditions and tests for task matching. In both cases, the DL knowledge base we are dealing with is quite large. Moreover in the case of state information, the DL KB is constantly changing as the planner simulates the effects of actions put into the partial plan generated during the search. The instances of the task ontology might also be

changing if the availability of services is changing, e.g. as in the discovery of new services. Query answering against changing KBs is even harder as the cached results are invalidated frequently.

For these reasons, the performance of the planning system is considerably affected by the query answering performance of the reasoner. The practicality of the planning system depends on how fast the conjunctive queries can be answered.

In this chapter, we present optimization techniques for conjunctive query answering. We examine boolean queries and retrieval queries separately. These techniques are applicable for query answering in general but we examine the cases that occur in HTN-DL in more detail.

We will start with the discussion of atomic queries, review some of the existing optimization algorithms, and describe how they interact with the optimization techniques described in the previous section. Then we look at boolean query answering and finally talk about answering conjunctive retrieval queries. We present how the reordering of query atoms can improve query answering time and present techniques and heuristics to find (near-)optimal orderings.

6.1 Answering Atomic Queries

We start with answering atomic ABox queries in the form $C(x)$ and $p(x, y)$. The case of ground and non-ground queries are examined separately.

6.1.1 Retrieving Instances

The ground query $C(a)$, so-called *instance check*, is answered by adding the negated statement $\neg C(a)$ to the ABox and checking for (in)consistency. KB consistency is an expensive operation so we want to avoid the consistency check as much as possible. We can use the completion graph created for the initial ABox consistency test. Since we always want to make sure that the information we have is consistent (any logical statement is entailed by an inconsistent KB) we will have the completion graph generated.As described in Section 5.3.3, caching this completion graph also improves the efficiency for subsequent reasoning steps.

The completion graph generated for the ABox can be used for both finding obvious instances and non-instances. If a concept exists in the label of a node with no dependency information then we can conclude that the concept will occur in every possible model of the KB and thus the individual is an instance. When we cache completion graphs, we also store the branching information, e.g. what kind of non-deterministic choices were made at each step and how many possibilities of that choice has been tried. For example, a disjunction branch for concept $C_1 \sqcup C_2 \sqcup C_3$ would say which of the disjuncts were already tried. Therefore, if C_3 is in the label of the node and we see that C_1 and C_2 had already been tried with no success, i.e the branches were closed with a clash, then we can again conclude that the individual corresponding to that node is an instance of C_3.

Note that, for *defined concepts*, even if the individual is an instance of the concept, checking the label of the node might not reveal this relation. For example,

consider the following definition of a bus driver:

$$BusDriver \equiv Driver \sqcap \exists drives.Bus$$

In a completion graph, the label of a node might not contain the concept $BusDriver$ but if it contains the concept $Driver$ and has an outgoing edge labeled with the role $drives$ (and none of the labels has a dependency) then we can conclude that the individual is an instance of $BusDriver$. Therefore, we can break up the concept definition into its components and check for obvious instances and non-instances for each component recursively. If the individual is an obvious instance of all the conjuncts then we say it is an obvious instance of the concept, if its an obvious non-instance of at least one conjunct then it is not an instance of the concept. The same idea can be applied to disjunctive definitions.

We can also use the completion graph for finding obvious instances of existential concepts. Let us illustrate this with an extension of the previous example. Suppose we have the following TBox axioms

$$Driver \equiv Person \sqcap \exists drives.Vehicle$$

$$Bus \sqsubseteq Vehicle$$

and the ABox assertions

$$Driver(\texttt{Bob}), drives(\texttt{Bob}, \texttt{Bus42}), Bus(\texttt{Bus42})$$

Now, even if we examine the parts of $BusDriver$ concept definition separately, we will not find $\exists drives.Bus$ in the label of the node corresponding to `Bob`. However, finding the $drives$ edge we can check the node for `Bus42` to see if it is an instance of Bus.

Obvious non-instances can be detected by the pseudo model merging technique explained in Section 5.4.1 without performing a consistency test [47]. If we have a pseudo-model for the negation of the concept (which can be built after a satisfiability test) we check if the pseudo-model of the concept can be merged with the pseudo-model of the individual (we can simply reuse the node from ABox completion as the pseudo-model of the individual). If there are no interactions between two nodes that could possibly cause a clash we conclude that the individual is not an instance of the concept.

The naive way to answer the non-ground atomic query $C(x)$, so-called *instance retrieval*, is to iterate over all the individuals in the ABox and do a consistency check when the above methods fail to detect an obvious instance or non-instance. Typically most of the remaining individuals are non-instances and do not cause an inconsistency when the negated statement is added to the KB. *Binary instance retrieval* technique presented in [48] exploits this characteristic and combines many instance checks in one ABox consistency test. If there is no inconsistency all the candidate individuals are proven to be non-instances, otherwise the method splits the set of candidates into two and continues. Clearly, the effectiveness of binary instance retrieval is maximized if the candidate list contains less instances.

Algorithm 12 shows the pseudo-code that combines all of the mentioned tech-

niques. Note that, we have a special case for concepts in the form $\exists p.D$ but not $\forall p.D$. This is simply because having one model of the KB where the $\forall$ restriction is satisfied is not enough evidence to conclude for instance relation.

6.1.2 Retrieving Role Fillers

In OWL-DL, the role constructors are much less expressive compared to concept constructors. In less expressive fragments, the relation $p(\mathtt{a}, \mathtt{b})$ holds only if in the original ABox it is asserted that a and b is related by p or one of its subroles. The interactions between the role hierarchy and number restrictions invalidate this assumption, e.g. a super role assertion combined with cardinality restrictions may cause the relation to hold. Transitive roles complicate the situation even more; now a path between a and b is enough for the relation to hold. Having nominals in the KB makes things even more complicated as nominals might relate individuals from disconnected parts of the ABox.

This observation might lead to think that we will need to check for every possible pair of individuals to find all the tuples in the $p(x, y)$ relation. Fortunately, this is not the case. Instead of examining the asserted facts in the ABox, we can inspect the completion graph generated for the ABox consistency test. Suppose in this completion graph, there is an edge between a and b labeled by the role p or one of its subroles. If this edge does not depend on any non-deterministic choice then we can conclude that $p(\mathtt{a}, \mathtt{b})$ is entailed. As we have shown in the nominal-based model merging technique of Section 5.4.1, if no such edge exists then the relation is

Algorithm 12 $isKnownInstance(\mathcal{K}, \mathbf{G}, n, C, S)$

Inputs: $\mathcal{K} = \langle \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ is the input KB, $\mathbf{G}$ is the complete clash-free graph for $\mathcal{A}$, n is a node in $\mathbf{G}$ corresponding to an individual i, C is a (possibly complex) concept, S is the set of seen $\langle n, C \rangle$ pairs to avoid infinite cycles

Outputs: Returns `true` if i is obviously a C instance, `false` if it is an obvious non-instance, `unknown` if neither can be proven without a consistency test

Let $result =$ `unknown`
if $\langle n, C \rangle \in S$ **then**
  return $result$
else
  Let $S = S \cup \langle n, C \rangle$
end if
if $C \in \mathcal{L}(n)$ with empty dependency set **then**
  Let $result =$ `true`
else if mergable(n, $\neg C$) **then**
  Let $result =$ `false`
else if $(C \equiv D) \in \mathcal{T}$ **then**
  Let $result = isKnownInstance(\mathcal{K}, \mathbf{G}, n, D, S)$
else if C is in the form $C_1 \sqcap \ldots \sqcap C_k$ **then**
  Let $result =$ `true`
  for all $i : 1 \leq i \leq k$ **do**
    Let $result_i = isKnownInstance(\mathcal{K}, \mathbf{G}, n, C_k, S)$
    if $result_i =$ `false` **then**
      Let $result =$ `false` and **exit loop**
    else if $result_i =$ `unknown` **then**
      Let $result =$ `unknown`
    end if
  end for
else if C is in the form $C_1 \sqcup \ldots \sqcup C_k$ **then**
  for all $i : 1 \leq i \leq k$ **do**
    Let $result_i = isKnownInstance(\mathcal{K}, \mathbf{G}, n, C_i, S)$
    if $result_i =$ `true` **then**
      Let $result =$ `true` and **exit loop**
    end if
  end for
else if C is in the form $\exists p.D$ **then**
  if n has no p-neighbor **then**
    Let $result =$ `false`
  else if n has a p-neighbor m with no dependency **then**
    Let $result = isKnownInstance(\mathcal{K}, \mathbf{G}, m, D, S)$
  end if
end if
return $result$

not entailed because there is at least one model where the relation does not hold.

If there is an edge between $\mathtt{a}$ and $\mathtt{b}$ but the edge depends on a non-deterministic choice, we cannot conclude if the relation holds or not. For example, if we have the the assertion $(\exists p.\{b\} \sqcup C)(a)$ in the KB $\mathcal{K}$ then we might end up with such a completion graph. If $\mathcal{K} \cup \neg C(a)$ is inconsistent then $p(\mathtt{a}, \mathtt{b})$ holds, otherwise it does not. All such individuals are *possible* role fillers for individual $\mathtt{a}$.

If we have some possible candidates as role fillers, we can reduce the query $p(x, \mathtt{a})$ (resp. $p(\mathtt{a}, x)$) to an instance retrieval query for concept $\exists p.\{a\}$ (resp. $\exists p^{-}.\{a\}$).

If both arguments in the query are non-ground as in $p(x, y)$, then we first need to generate all candidates for x (e.g. by retrieving the instances of $\exists p.\top$) and then use the above techniques to find corresponding y values.

6.2 Answering Conjunctive Boolean Queries

As we have discussed in Section 3.3.1, the rolling-up technique can be used to answer boolean conjunctive queries if there is no cycle involving only variables. The query Q is rolled-up into a concept expression C_Q and we test for KB satisfiability after adding the axiom $C_Q \sqsubseteq \bot$.

One immediate observation is that such a consistency test will be quite expensive since we cannot reuse the cached completion graph due to the additional axiom in the TBox. However, if we have a constant mentioned in the query we can equivalently reduce the problem to instance checking. Suppose we have the

following boolean query

$$Q(x, y) \rightarrow C(x) \wedge p(x, \mathtt{a}) \wedge q(\mathtt{a}, y) \wedge D(y)$$

We can select x, y, or the constant $\mathtt{a}$ as the root node for rolling up. If we choose $\mathtt{a}$ and proceed, the rolled-up concept we get is

$$C_Q = \{\mathtt{a}\} \sqcap \exists p^{-}.C \sqcap \exists q.D$$

It is very straight-forward to prove that $\mathcal{K} \cup \{\top \sqsubseteq \neg C_Q\}$ is inconsistent iff $\mathcal{K} \models (\exists p^{-}.C \sqcap \exists q.D)(\mathtt{a})$ (direct consequence of the proof of Proposition 5.1) .

With this equivalence we can simply use instance checking and take advantage of the optimization techniques we presented in the previous section. It is interesting to note that concepts we generate by rolling-up primarily consist of conjunctions and existential restrictions (of course other type of complex expressions might be directly used in the query). Therefore, the Algorithm 12 can be effectively used to return obvious answers without a consistency test.

Recall that, boolean queries in HTN-DL e.g. precondition of an operator, is evaluated only after we have the input bindings for the operator. For this reason, the query expressions always have a constant value and we can use instance checking to answer the query.

6.3 Answering Conjunctive Retrieval Queries

Answering retrieval queries can be done by assigning an individual to each distinguished variable to obtain a (partially-)ground boolean query and then checking the if the boolean query is entailed. If the boolean query is entailed, the tuple used in the assignment will be in the answer set of the retrieval query.

Trying all possible tuples in the domain is obviously not practical. As suggested in [65] one can first roll-up the query without any assignment and retrieve possible candidates for each variable making the search space smaller. There is still one important drawback of this approach which stems from not having the ability to see why a particular binding fails. Suppose we have a query $p(x, y) \wedge q(y, z)$ with three distinguished variables and we have 10 possible candidates for each variable. If the individuals x_1 and $y1$ (the first candidates for variables x and y respectively) are not related with p then regardless of any assignment to z any tuple having x_1 and y_1 fail. If we build tuples incrementally by checking the satisfiability at each step we can eliminate many possibilities early.

The retrieval queries we encounter in HTN-DL are nearly always queries with only distinguished variables. Recall that, the local variables of a method act as the distinguished variables of a query and later they are used as input values to subtasks. For this reason, we will focus on answering queries with only distinguished variables.

If there are only distinguished variables in a query then we do not need the rolling-up technique to answer the query. As we explained at the beginning of the chapter, instance retrieval is most effective with named concepts. If we use

rolling-up to generate complex concepts, we would later break up the concept to its components to find obvious instances and non-instances. Repeating this step for every possible candidate is wasteful. Instead, we can consider each atom separately which also makes it easier to generate tuples incrementally.

Algorithm 13 presents a query answering algorithm for queries with only distinguished variables. The algorithm simply iterates through all the atoms in the query and either generates bindings for a variable or tests if the previous bindings satisfy the query atom. Generating bindings are done by invoking the instance retrieval function *retrieve* which in turn might perform several consistency checks as described earlier. Theoretically, testing the satisfaction of a query atom might also require a consistency check. However, as explained in the previous section, most of these tests can be answered without doing a consistency test. Especially, the retrieval operations regarding the role assertions, e.g. $retrieve(\exists p.\{\mathtt{u}\})$, do not typically require any consistency check.

Initially the algorithm is invoked by $AnswerQuery(\mathcal{K}, A, \emptyset, \emptyset)$ where A is an ordering of the atoms in the query. The correctness of this algorithm is quite clear as the satisfaction of every binding is reduced to KB entailment. Thus, this is a sound and complete procedure for answering conjunctive queries.

6.4 Cost-based Query Reordering

The efficiency of Algorithm 13 depends very much on the order query atoms are processed. For example, in the query $C(x) \wedge\ p(x, y) \wedge\ D(y)$, suppose C has

Algorithm 13 $AnswerQuery(\mathcal{K}, A, B, Sol)$

Inputs: $\mathcal{K}$ is the input KB, A is a list of query atoms, B is the binding built so far, Sol is the set of all bindings that satisfy the query

```
if A = [ ] then
  return Sol ∪ {B}
end if
Let a = first(A) and R = rest(A)
Substitute the variables in a based on the bindings in B
if a = C(v) and K |= C(v) then
  Let Sol = AnswerQuery(K, R, B, Sol)
else if a = C(x) then
  for all v ∈ retrieve(C) do
    Let Sol = AnswerQuery(R, B ∪ {x ← v}, Sol)
  end for
else if a = p(v, u) and K |= p(v, u) then
  return AnswerQuery(K, R, B, Sol)
else if a = p(x, v) (resp. p(v, x)) then
  for all u ∈ retrieve(∃p.{v}) (resp. u ∈ retrieve(∃p⁻.{v})) do
    Let Sol = AnswerQuery(K, R, B ∪ {x ← u}, Sol)
  end for
else if a = p(x, y) then
  for all v ∈ retrieve(∃p.⊤) do
    for all u retrieve(∃p.{v}) do
      Let Sol = AnswerQuery(R, B ∪ {x ← v} ∪ {y ← u}, Sol)
    end for
  end for
end if
return Sol
```

100 instances, each instance has one p value and D has 10.000 instances. The ordering $[C(x), p(x, y), D(y)]$ would be much more efficient compared to the ordering $[D(x), p(x, y), C(y)]$. We would do one instance retrieval operation to get 100 instances, find the corresponding p values and test whether these are D instances. The second ordering, on the other hand, requires us to iterate over 10.000 individuals

and check for a p^{-} value that does not exists for most d instances.

There are several important challenges to finding an optimal query reordering. In query optimization for relational databases, the main objective is to find an optimal join order and generally the bottleneck is reading data from disk. In a DL reasoner, the most costly operation is consistency checking so we should try to minimize the number of consistency checks performed.

There are two parameters that will help us to estimate the cost of answering a query. First, we need to estimate how costly an atomic query is, e.g. for an instance retrieval query, estimate how long it will take to find all the instances. Second, we need to estimate the size of the results, e.g. how many instances a concept has. These two parameters are interdependent to some degree. For example, if C has 100.000 instances and D has only 10 instances, retrieving C instances is typically more costly. However, this is not always true because there might be 100.000 individuals that are considered as possible D instances (i.e. the methods described in Section 6.1 failed for all those individuals). In that case, we would be forced to do many expensive consistency tests. For this reason, there is no easy way of estimating these parameters that would work for in different situations. In the next section, we will describe some methods for computing these parameters.

For now, we will assume that for each atomic query type, there are cost functions $\mathcal{C}_{ir}(C)$, $\mathcal{C}_{ic}(C)$, $\mathcal{C}_{rr}(r)$ and $\mathcal{C}_{rc}(r)$ that returns the cost of instance retrieval for concept C, the cost of a single instance checking for concept C, the cost of role filler retrieval for role r and the cost of verifying a role filler for role r, respectively. Note that, we are assuming the cost of instance checking for a concept is the same for

all the different individuals in the KB. This assumption may not be very accurate but considering that we typically deal with large number of individuals, it is not practical to compute estimates for every individual.

In addition, we will assume that the size estimates for concepts and properties are also ready. We will use $|C|$ to denote the number of C instances and $|p|$ to denote the total number of tuples in p relation. The average number of p fillers for an individual is denoted by $avg(p)$ and computed as $|p|/|\exists p.\top|$. Similarly, we say $avg(p^-) = |p|/|\exists p^-.\top|$.

Given the parameters for the cost computation and the size estimates, Algorithm 14 computes an estimate for the cost of query answering for a certain ordering.

Cost estimation is linear in the number of query atoms, provided that size estimates are already computed. However, there are exponentially many orderings to try so an exhaustive search to find the best ordering is still very expensive. It is possible to use some heuristics to prune the search space. The heuristics we use are:

1. For each atom at position $i > 1$ in the ordered list, there should be at least one atom at position $j < i$ s.t. two atoms share at least one variable.

2. Atoms of the form $p(x, \mathtt{v})$ and $p(\mathtt{v}, x)$ should appear before other atoms involving x.

3. An atom of the form $C(x)$ should come immediately after the first atom that contains x.

The first rule is similar to the general query optimization rule that cross products should be avoided. The second rule makes use of the fact that generally an

Algorithm 14 *EstimateCost*(A, B)

Inputs: A is a sorted list of query atoms, B is the variables bound so far

if $A = [\,]$ **then**
  return 1
end if
Let $a = \text{first}(A)$ and $R = \text{rest}(A)$
if $a = C(x)$ and $x \in B$ **then**
  return $\mathcal{C}_{ic}(C) + EstimateCost(R, B)$
end if
if $a = C(x)$ and $x \notin B$ **then**
  return $\mathcal{C}_{ir}(C) + |C| * EstimateCost(R, B \cup \{x\})$
end if
if $a = p(x, y)$ and $\{x, y\} \subseteq B$ **then**
  return $\mathcal{C}_{rc}(p) + EstimateCost(R, B)$
end if
if $a = p(x, y)$ and $\{x, y\} \cap B = \{x\}$ **then**
  return $\mathcal{C}_{rr}(p) + avg(p) * EstimateCost(R, B \cup \{y\})$
end if
if $a = p(x, y)$ and $\{x, y\} \cap B = \{y\}$ **then**
  return $\mathcal{C}_{rr}(p) + avg(p^{-}) * EstimateCost(R, B \cup \{x\})$
end if
if $a = p(x, y)$ and $\{x, y\} \cap B = \emptyset$ **then**
  return $\mathcal{C}_{ir}(\exists p.\top) + |p| * \mathcal{C}_{rr}(p) * EstimateCost(R, B \cup \{x, y\})$
end if

individual is related to limited number of other individuals. The last rule is to discard the orderings such as $[C(x), p(x, y), q(y, z), D(y)]$. This ordering is not desirable because if $p(x, y)$ finds a binding for y such that $D(y)$ is not satisfied, we would unnecessarily retrieve the q fillers before realizing the failure.

6.4.1 Size and Cost Estimation

There are several different ways to estimate $|C|$. Of course we do not want to perform any consistency test to estimate the size as this would defeat the purpose of computing the size estimate. The most straight-forward way is to examine the asserted facts in the ABox to figure out which individuals are obvious instances. Examining the completion graph of the ABox will give a better estimate. We can use the more advanced technique of Section 6.1 to obtain even more accurate estimates.

The main idea behind size estimation is to iterate over the nodes in the completion graph of ABox and for each concept call the algorithm $isKnownInstance$. The algorithm might return `true`, `false`, or `unknown`. It is not possible to know (without a consistency test) how many of the individuals with `unknown` result will end up being instances. In such cases, we estimate that with probability κ such individuals would indeed be instances of that concept giving us the formula $|C| = |C_{known}| + \kappa * |C_{unknown}|$.

This technique will give a quite accurate estimate on the size of instances. However, as the number of instances increases, iterating over all the individuals and concepts would be quite time-consuming and not practical even as a preprocessing step. One observation is that we do not need to iterate over all the concepts as some of them will not be used in any of the queries. For an HTN-DL domain, we could inspect the precondition expressions and determine which concepts are mentioned in the queries. Or alternatively we can generate these statistics on-the-fly when a query arrives and compute the size estimation only for the concepts mentioned in

the query.

A more effective solution is to use random sampling which has proven to be very useful in relational database settings. We can simply select a random sample of individuals from the ABox and compute the estimates based on that smaller estimate. If σ is the sampling ratio, the size estimation formula would be $|C| = \sigma * (|C_{known}| + \kappa * |C_{unknown}|)$.

We have not talked about size estimation for roles but the same principals and algorithms can be directly used for roles, too. As we are iterating over the individuals, we can use the techniques of Section 6.1.2 to find obvious and possible role fillers and compute the size estimate for roles.

We can also use $|C_{unknown}|$ to have an estimate about $\mathcal{C}_{ir}(C)$ and $\mathcal{C}_{ic}(C)$. If $|C_{unknown}| = 0$ then it means that all the individuals can be retrieved without any consistency test. As $|C_{unknown}|$ increases $\mathcal{C}_{ir}(C)$ will increase because more consistency tests will be needed.

6.5 Query Simplification

In some cases, there might be redundant atoms in a query that can be safely removed from the query without affecting the results. For example, if $C \sqsubseteq D$ then the query $C(x) \wedge D(x)$ is logically equivalent to query $C(x)$. Such redundant atoms do not cause to make additional consistency tests (methods described in Section 6.1 are quite effective for these cases) but even repeating computationally cheap operations many times causes a noticeable overhead in the end.

The idea behind query simplification is to discover redundant atoms with cheap concept satisfiability tests. But performing too many concept satisfiability tests for simplifications that do not occur frequently in queries is wasteful. For example, simplification based on subsumption of named concepts and roles are nearly never applicable in real world queries or in the benchmarking problems for query answering.

We have pinpointed the following two common query simplifications based on domain and range restrictions. If the DL conjunctive query contains the following set of atoms

$$Q = \{C(x), p(x, y), D(y)\}$$

then we can simplify it in two different ways as follows

$$Q' = Q \setminus \{C(x)\} \text{ if } \exists p.\top \sqsubseteq C \quad (\textit{Domain simplification})$$

$$Q'' = Q \setminus \{D(y)\} \text{ if } C \sqsubseteq \forall p.D \quad (\textit{Range simplification})$$

Note that domain/range simplification can also be done even if one of the atoms $C(x)$ or $D(y)$ is missing since we can simply insert $\top(x)$ or $\top(y)$ as an additional atom. In such cases global domain/range restrictions of properties can be directly used and simplification can be done with no subsumption test.

Chapter 7

Implementation and Evaluation

In this chapter, we describe the HTN-DL planning system and its components: Pellet OWL-DL reasoner, OWL-S Web Services API, and HTN-DL planner. We describe the architecture of each component, discuss their role and integration in the HTN-DL planning system and also explain their impact outside HTN-DL. We also present performance evaluation results about the reasoner and planner showing the effectiveness of the optimization techniques presented and the practicality of HTN-DL system for Web Service composition composition problems.

7.1 System Architecture

Figure 7.1 shows the main components of the HTN-DL planning system. The implementation of the planning algorithm described in Section 3.3 is responsible of solving HTN-DL planning problems. HTN-DL planning domains are created by the translation algorithm of Section 4.2 which is implemented on top of OWL-S API. Execution of Web Services to gather information is also done using the execution engine of OWL-S API.

The composition process starts by translating OWL-S descriptions to HTN-DL domains. HTN-DL system does not perform Web Service discovery; that is, it assumes all the service descriptions (regardless of what functionality they provide) are

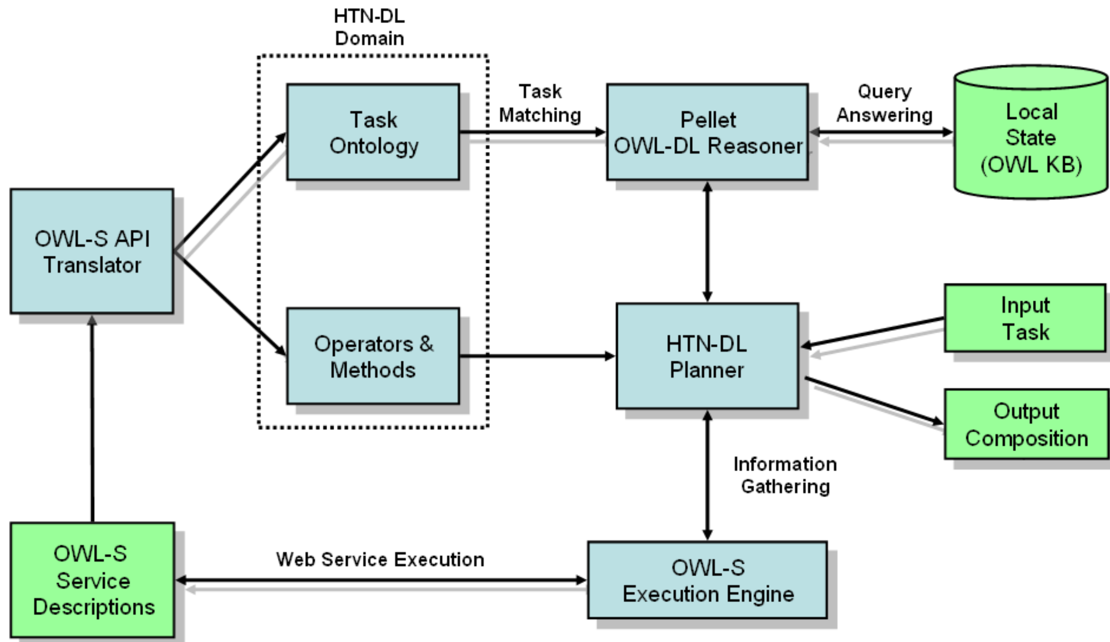

Figure 7.1: Overview of HTN-DL planning system

given to the system as input. HTN-DL domain generated contains both a task ontology (translation of OWL-S profile descriptions) and a set of method and operator definitions (translation of OWL-S process models).

The goal of the composition is given to the planning system as a partially-ordered ground task network. The state of the world is represented as a DL knowledge base. HTN-DL planner uses the reasoner Pellet to find the matching operators and methods for a given task. Then the planner, again using the reasoner, evaluates the preconditions of actions to determine applicability.

7.2 Pellet: OWL-DL Reasoner

Pellet reasoner is the main driving force behind the HTN-DL planning system. The reasoner is responsible of handling the maintenance of state including precondition evaluation and effect application and also matching tasks with operators and methods. Besides its functionality in HTN-DL Pellet is a full-fledged OWL-DL reasoner with many novel features. In this section, we describe Pellet's architecture, its main components and special features.

7.2.1 Pellet Architecture and Design

Pellet, in its core, is a Description Logic reasoner. However, unlike other DL reasoners, it has been designed to work with OWL right from the beginning. This design choice had huge influence on the overall architecture. It affected how the tableau reasoner was implemented, e.g. with the ability to reason with instance data (ABox reasoning) without making the Unique Name Assumption (UNA), and what kind of supporting modules to have, e.g. having an XML Schema datatype reasoner and a query engine.

Figure 7.2 shows the main components of Pellet. The core of the system is the tableaux reasoner that checks the consistency of a knowledge base. The reasoner is coupled with a datatype oracle that can check the consistency of conjunctions of (built-in or derived) XML Schema simple datatypes. The OWL ontologies are loaded into the reasoner after species validation and ontology repair. This step ensures that all the resources have an appropriate type triple (a requirement for

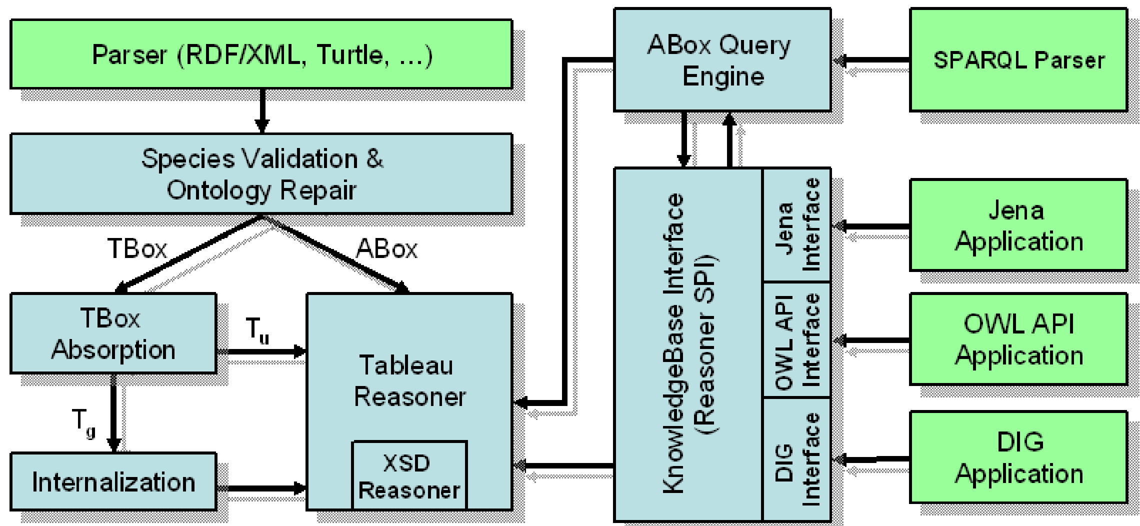

Figure 7.2: Main components of the Pellet reasoner

OWL-DL but not OWL-Full) and missing type declarations are added according to some heuristics (see subsection 7.2.3 for details). During the loading phase, axioms about classes are put into the TBox component and assertions about individuals are stored in the ABox component. TBox axioms go through the standard preprocessing of DL reasoners, e.g. normalization, absorption and internalization, before they are fed to the tableaux reasoner. The system provides a thin layer for programmatic access through the Service Programming Interface (SPI) that provides convenience functions to access the reasoning services provided.

7.2.2 Tableaux Reasoner

The tableaux reasoner has only one functionality: checking the consistency of an ontology. As explained in Section 2.3 all other reasoning tasks can be defined in terms of consistency checking. In order to support future extensions, the internals

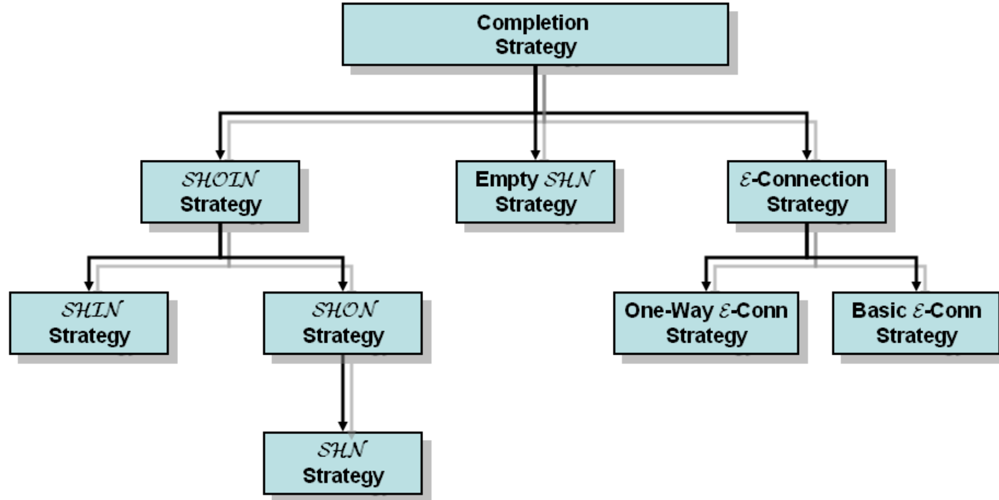

Figure 7.3: Different completion strategies implemented in Pellet

of the tableaux reasoner are built on an extensible architecture.

The completion algorithm inside the tableaux reasoner is designed so that different completion strategies can be plugged in. This approach has two major advantages: First, different completion strategies with different heuristics can be used based on the characteristics of the given KB, e.g. the expressivity of the KB. Second extensions that need quite different completion strategies, e.g. $\mathcal{E}$-Connection reasoning, can be implemented without changing the rest of the system. Figure 7.3 shows the different completion strategies currently implemented in Pellet. $\mathcal{SHOIN}$*Strategy* is the default completion strategy that supports the full expressivity of OWL-DL. This strategy is based on the recently developed decision procedure for $\mathcal{SHOIQ}$[62].

The $\mathcal{SHOIN}$*Strategy* covers the full expressivity of OWL-DL and exhibits a good "pay as you go" behavior, e.g. the tableaux rule for nominals is never applied if there are no nominals in the KB. However, the blocking strategy required for

$\mathcal{SHOIN}$ is dynamic double blocking which is quite complex and may not prevent the completion graph from getting very large. If it is known that there are no nominals in the KB, e.g. the expressivity is $\mathcal{SHIN}$, then an optimized version of double blocking [61] can be used. Also, in this case, we do not even need to check if nominal rule is applicable (since it will never be) and save some more time. The *$\mathcal{SHIN}$Strategy* does exactly this, and hence, whenever the expressivity of the KB is detected to fall into this category, this strategy will be selected over the default *$\mathcal{SHOIN}$Strategy*. Similarly, the *$\mathcal{SHON}$Strategy* employs an even more efficient blocking strategy (subset blocking) [56] and is selected whenever appropriate.

Some completion strategies behave quite different than others. For example, if there are no instances in the KB (just class and property descriptions) then it is known that every concept satisfiability check will start with a completion graph that has just one node. If there are also no inverse roles in the KB, more efficient completion strategies, e.g. the trace method, can be used. In such case, we can use additional optimizations such as caching the satisfiability status of internal nodes. The *Empty$\mathcal{SHN}$Strategy* uses this approach and manages to handle large TBoxes such as the famous Galen medical ontology.

The dynamic completion strategy selection ensures the soundness and completeness of the reasoner (for each strategy we use only optimization techniques that are known to be sound and complete) while exploiting the most efficient algorithm for the given KB.

7.2.3 OWL Species Coercion

OWL ontologies are encoded as RDF/XML graphs. OWL-DL imposes a number of restrictions on RDF graphs, some of which are substantial (e.g., that the set of class names and individual names be disjoint) and some less so (that every item have an `rdf:type` triple). Ensuring that an RDF/XML document meets all the restrictions is a relatively difficult task for authors, and many existing OWL documents are nominally OWL-Full, even though their authors intend for them to be OWL-DL. Pellet incorporates a number of heuristics to detect "DLizable" OWL-Full documents in order to "repair" them.

The heuristics implemented in Pellet attempt to guess the correct type for an untyped resource. These are mainly standard operations, e.g. a resource used in the predicate position is inferred to be a property. Some situations have more than one solution, e.g. an untyped resource used only in one cardinality restriction can be any of object or a data property. In these cases, Pellet heuristics choose object properties and classes over data properties and datatypes by default, but this behavior can be configured.

Ensuring the vocabulary separation, e.g. disjointness of classes, properties and individuals, is another hard problem especially in the distributed Web environment where people might be required to import an OWL-Full ontology that they might have no control over. In such a case, it is not acceptable for a reasoner to reject processing the ontology altogether. For this reason, Pellet provides several options to the users where vocabulary separation is not respected:

- Ignore the statements that cause the problem. If a URI is used both as a class and as a property, one of these definitions will be ignored and the accepted definition depends on the order the statements are processed (this order is generally non-deterministic and based on which underlying parser is used).

- Accept all the definitions for the URI but treat them differently for query answering. For example, if the same URI is defined both as a class and as a property, Pellet will create both a class and a property and associate the axioms with the corresponding definition. Depending on the queries, asking subclasses vs. asking sub properties, the appropriate definition will be used.

- Reject processing the ontology completely.

These options give the user more control about how to deal with the different cases and provide a plausible solution for a certain set of OWL-Full ontologies. On the other hand, some features of OWL-Full ontologies are completely out of scope for Pellet. For example, defining cardinality restrictions on transitive properties causes undecidability. Extending built-in vocabulary, e.g. creating a subproperty of `rdf:type`, requires a completely different reasoning procedure. Therefore, for such OWL-Full features only options provided are *Ignore* or *Fail*.

7.2.4 ABox Query Engine

Pellet ABox query engine is based on the query answering algorithms and optimizations described in Chapter 6. Figure 7.4 shows the general design of the query engine. The query engine is composed of several modules. Initially, the query

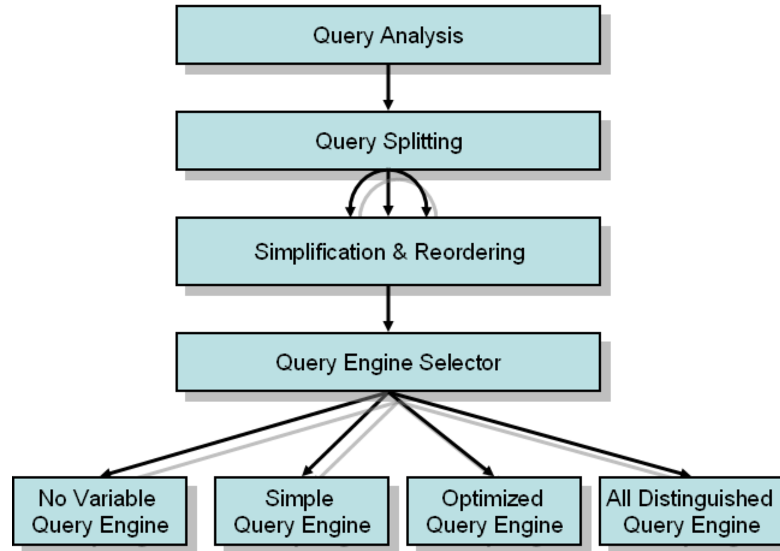

Figure 7.4: Components of the query engine

goes through several preprocessing steps. The first step of is to analyze the query and determine if it consists of independent sub-queries, that is, query graph is composed of disconnected graphs. If this is the case, the query is split into multiple queries which are answered separately. The results are combined at the end on a tuple by tuple basis to minimize memory consumption.

After this initial step, the query is modified based on the optimization techniques described in Chapter 6. The actual query answering is done by one of several different query answering engines. Each query is answered by exactly one engine and the selection is based on the properties of the query, e.g. whether it is a boolean query or a retrieval query, or how many distinguished/non-distinguished variables exist. Query engine selection and query optimization is not completely separate. For example, if there is a single variable in the query then reordering the query atoms does not make sense. So these two steps are mixed together.

7.2.5 Special Features

The extensible architecture of Pellet made it possible to develop some special features that do not exist in any other reasoners. Here, we briefly describe these features:

Axiom pinpointing Axiom pinpointing is a non-standard DL inference service that provides a *justification* for any arbitrary entailment derived by a reasoner from an OWL-DL knowledge base. Given a KB and any of its logical consequences, the axiom pinpointing service determines the premises in the KB that are sufficient for the entailment to hold. The justification is useful for understanding the output of the reasoner, which is key for many tasks, such as ontology debugging, design and evolution.

In order to determine a justification for an entailment, Pellet tracks and stores the original source axioms from the ontology as they are modified and used throughout the tableaux expansion process. For this purpose, we extend the dependency sets used in the clash detection procedure to keep track of the axioms. As the reasoner continues applying the tableau rules, the axiom set for each assertion needs to be updated as well as the dependency set information. When an *inconsistency-revealing* clash is discovered, the axiom set is presented along with the clash information. This ensures that only the axioms directly relevant to the inconsistency are obtained.

In order to determine all the justifications, Pellet uses a combination of tracing and a variant of Reiter's hitting set algorithm (see [70]). Single justification tracing involves almost no overhead and the results are used for generating on-demand

explanations and for improving the efficiency of incremental reasoning through updates.

$\mathcal{E}$-Connections $\mathcal{E}$-Connections [76] are a framework for combining several families of decidable logics, such as Description Logics, Modal Logics, as well as some logics of time and space. In an $\mathcal{E}$-Connection, the coupling between the combined logics is loose enough for obtaining general results about transfer of decidability: if reasoning is decidable in each of the component logics, then it is decidable in the combined formalism as well.

In [44] we have proposed tableau algorithms for different $\mathcal{E}$-Connection languages involving Description Logics. The basic strategy to extend a DL tableau algorithm with $\mathcal{E}$-Connections support is based on "coloring" the completion graph. Nodes of different "colors", or sorts, correspond to different domains (ontologies). The application of the expansion rules, blocking conditions and clash triggers depend on both the "color" of the node under consideration and the expressivity allowed on the link relations. (For a detailed discussion on combined tableau algorithms for $\mathcal{E}$-Connections we refer the reader to [44].) Pellet has been extended with tableau-based decision procedures for $\mathcal{E}$-Connection languages involving combinations of $\mathcal{SHOIN}(\mathcal{D})$ ontologies. The initial experimental results show that the performance for the $\mathcal{E}$-Connected KBs is very similar to their OWL counterparts.

Rules Pellet has support for $\mathcal{AL}$-log[25] (Datalog + $\mathcal{SHOIN}(\mathcal{D})$) via a coupling with a Datalog reasoner. It incorporates the traditional algorithm (described in

[25]) and a new precompilation technique that is incomplete but more efficient. The key idea of this implementation is to pre-process all of the DL atoms that appear in the Datalog rules, and include them as facts in the Datalog subsystem. Once the pre-processing is done, queries can be answered by the Datalog component using any of the known techniques for Datalog query evaluation.

Pellet also has a preliminary implementation of a direct tableau algorithm for a DL-safe rules [91] extension to $\mathcal{SHOIN}(\mathcal{D})$. Preliminary empirical results have been encouraging and we think that the DL-safe implementation is practical for small to mid-sized ontologies esp. when the full expressivity of $\mathcal{SHOIN}(\mathcal{D})$ is needed.

7.3 OWL-S API: API for Web Service

The OWL-S service descriptions are simply OWL-DL ontologies mostly containing instances of concepts defined in OWL-S ontologies. As such, service descriptions are canonically expressed in the OWL using RDF/XML exchange syntax. There are many tools that work with an RDF based model but working with OWL-S descriptions at the RDF or even the OWL level is quite difficult and tedious as they tend to be at the wrong level of abstraction. Furthermore, the OWL-DL axioms do not sufficiently constrain the OWL-S descriptions as we have discussed in Chapter 4. So, for programmatic generation of descriptions, for validation, for certain sorts of reasoning and for execution and monitoring, it is helpful to have service descriptions represented at a higher level of abstraction.

The OWL-S API is a Java library which provides this higher level of abstraction. These classes also support various useful services such as validation of OWL-S descriptions (beyond what is expressed in the ontologies), matchmaking, and execution.

7.3.1 The Design Objectives

The OWL-S API was designed to let programmers access and manipulate OWL-S service descriptions easily. For this reason, the main purpose of the API is to provide a data model that covers the specifics of OWL-S. However, the design of the API was driven by many other factors and objectives:

Support for multiple OWL-S versions OWL-S ontologies are constantly being refined and extended by the OWL-S coalition. The radical changes in the ontologies between different versions make it harder to develop and maintain applications based on the structure of the OWL ontologies. For example, the OWL-S processes were modeled as OWL classes in OWL-S 0.9 whereas they are modeled as OWL individuals in OWL-S 1.0 and higher versions. The data model in the API should be general enough to support different versions of the ontologies.

Execution of services OWL-S Process Model defines how a service works. Processes are defined either as one-step directly-invocable *AtomicProcess*es or as *CompositeProcess*es that are composed of other processes combined with one (or more) of the control construct defined in the Process ontology. An execution engine should

handle interpreting these control constructs. The invocation of *AtomicProcess*es are described by *Grounding* specifications that map the processes to WSDL operations. Some applications may extend the grounding specification to use other standards such as UPnP. The execution engine should support the invocation of WSDL services as a minimum requirement but it must also be flexible to handle other grounding specifications.

Extensibility of OWL-S descriptions One essential feature in describing Web Services with OWL-S is being able to extend the base OWL-S ontologies in order to describe specific features for a service. For this purpose, OWL-S profile ontology defines a construct named *ServiceParameter* so that concepts defined in other ontologies may easily be integrated into OWL-S profile descriptions. The API should let the users easily handle the concepts that are not part of the core OWL-S ontologies, thus not part of the core data model in the API.

7.3.2 Architecture of the OWL-S API

The OWL-S API was designed to achieve the objectives described in the previous section. The data model for services were created to reflect the structure of the OWL-S model. While the Java interfaces and methods were designed in conjunction with the classes and properties defined in the OWL-S ontologies, there is no tight coupling between the two. A set of *Reader*s have been created to parse descriptions of different versions of OWL-S into the same data model. Therefore, there is one consistent view for all the services even if different versions of the ontologies have

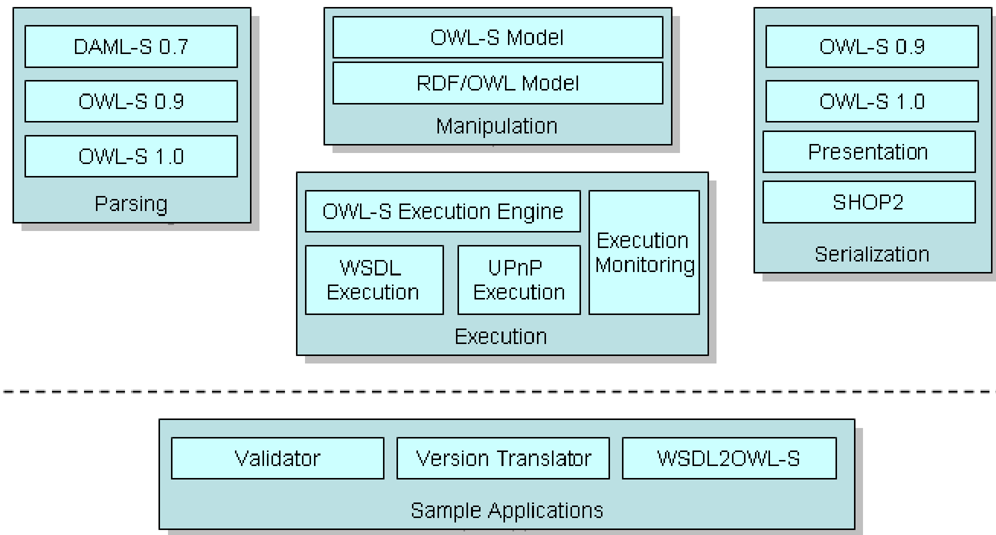

Figure 7.5: Basic components of the OWL-S API

been used. Serialization of the descriptions are handled by a set of *Writer*s. *Writer*s are generally used to serialize the service for a specific OWL-S version but may also be used for other purposes, e.g. generating an HTML presentation for the service. The basic components of the API are shown in the Figure 1.

The API has been built using the Jena [17] toolkit but the interfaces has been designed so that functionality of the OWL-S API is not bound to the specifics of the underlying RDF/OWL API. A basic *OWLResource* interface is provided for accessing the information in the RDF model and can easily be implemented for different RDF toolkits. Querying the RDF model makes it possible to get the extended parameters that are not part of the standard OWL-S ontologies. It is also possible to wrap the frequently used OWL concepts in a Java interface. This feature is similar to polymorphic views in the Jena toolkit and makes programming easier when applications are being developed for a fixed set of ontologies. Creation of Java

interfaces can even be automated so a code template can be generated for a given OWL-S description, similar to how stubs are generated from WSDL descriptions.

The execution of processes are handled by a *ProcessExecutionEngine*. The default implementation provides the execution of all control constructs defined in OWL-S. The OWL-S API is coupled with the Pellet reasoner which is used to verify the preconditions of services before execution and evaluate the conditions expressed in control constructs such as *If-Then-Else*, *Repeat-While*, and *Repeat-Until*.

The invocation of WSDL services are achieved through the Axis Web Services package. The API can also execute OWL-S services that have UPnP groundings through the extensions developed in collaboration with Fujitsu Labs of America, College Park in the context of Task Computing [84] for interacting with devices in pervasive environments.

7.4 HTN-DL: Planning for Web Services

7.5 Experimental Evaluation

In this section, we present the evaluation of the HTN-DL planning system. We start with the evaluation of reasoner Pellet in isolation. We show that optimization techniques presented in this thesis improve reasoning performance in general. Then we examine the performance of the coupling in the HTN-DL system. We test the planning system both on examples coming from standard planning benchmark problems and also on Web Service composition problems,

All the experiments presented in this section have been performed on a Pen-

tium Centrino 1.6GHz computer with 1.5GB memory using Java 1.4.2. The experiment shave been repeated 20 times and averages are presented.

7.5.1 Reasoning Performance

We start with the performance evaluation for the tasks of consistency checking, classification and realization. Then we show the results on conjunctive query answering.

Evaluating Reasoning Optimizations

We have run the experiments on four ontologies: the Wine ontology, presented in the OWL documentation [112], the AKT Portal Ontology, used in the AKT project for integrating information across universities, the OWL-S ontologies, for describing Web Services, and the 3SAT ontology, included in the OWL test suite, which is an encoding of the classical 3SAT problem in OWL-DL.

In order to evaluate the impact of each optimization, we have disabled the optimizations one by one when processing each ontology. The results are shown in Table 7.1. The first column indicates the enabled optimizations; the remaining columns show the times for the initial ontology consistency check, classification (including satisfiability of atomic concepts) and realization of individuals respectively.

The Wine Ontology is a medium-size ontology and it uses all of the constructs provided in OWL-DL. It contains 137 atomic concepts, 17 roles and 206 individuals. The concepts defined in the ontology are fairly complex and nominals are used

	Wine			OWL-S		
Options	Consist.	Classif.	Real.	Consist.	Classif.	Real.
OHDMLC	772.0	16911.4	2154.3	377.6	2422.5	1021.5
_HDMLC	16608.9	N/A	N/A	407.6	2634.7	1141.8
O_DMLC	21748.2	64463.7	61412.4	387.4	2500.7	1062.5
__DMLC	230463.5	N/A	N/A	388.7	2488.4	1083.6
OH_MLC	3184.3	27182.1	35246.7	18006.8	2052.0	1059.5
OHD_LC	766.0	32294.3	9852.3	391.4	2461.7	1089.3
OHDM_C	779.1	20973.1	2155.4	387.3	45669.9	1113.4
OHDM__	793.2	N/A	N/A	389.4	72805.7	1116.6

	AKT Portal			3SAT		
Options	Consist.	Classif.	Real.	Consist.	Classif.	Real.
OHDMLC	6.0	399.6	47.0	1651.5	3.0	1.0
_HDMLC	7.0	2647.0	785.1	11478.5	3.0	6498.3
O_DMLC	2.0	374.6	41.1	1542.1	2.0	2.1
__DMLC	6.0	2606.7	786.3	8072.5	1.0	18493.7
OH_MLC	1.0	1607.2	49.1	1471.1	3.0	1.0
OHD_LC	3.0	382.6	43.2	920.5	1.0	0.0
OHDM_C	4.1	1030.4	44.1	1388.9	5.0	1.0
OHDM__	0.0	1503.3	42.0	1050.4	1362.0	0.0

Table 7.1: Consistency checking, classification and realization times for four different ontologies. All times are given in milliseconds. Classification times include concept satisfiability and subsumption tests. Realization time shows how long it took to find the most specific type for each individual. Each row gives the timing where a different set of optimizations is enabled. Each letter in the option description indicates which optimization is enabled, i.e. if there is no dash it means all the optimizations weer enabled. A dash indicates that the optimization reported in that position has been disabled. The letters used for the optimizations are as follows: Nominal absorption for OneOf (O) and hasValue (H), Learning-based Disjunct Selection (D), Nominal-based Pseudo-Model Merging (M), Lazy Completion Graph Generation (L), Completion Graph Caching (C).

profusely. With all the optimizations enabled, consistency checking takes less than a second, whereas the total processing time, including classification and realization takes approximately 20 seconds. Nominal absorption has the highest impact on performance: without any kind of nominal absorption Pellet cannot classify the ontology in the specified time limit and consistency time increases by three orders of magnitude.

Learning-based disjunct selection is especially effective for realization tests and nominal-based pseudo-model merging heavily influences classification, since it avoids a large number of subsumption tests. Lazy completion graph generation and graph caching have a dramatic impact on concept satisfiability and subsumption: if both optimizations are disabled, Pellet times out after the initial KB consistency test.

The OWL-S ontology is a medium-sized KB developed by the OWL-S coalition and widely used by the Semantic Web Services community. It contains 97 concepts, 191 roles and 2320 individuals, with 5 nominals. The individuals for our experiments represent Web services and have been generated in a realistic Task Computing environment [84] developed at Fujitsu Labs of America. OWL-S does use nominals, but marginally. The optimization with the most impact is disjunct selection, which makes it possible to identify similarity patterns between individuals and use them for making the right non-deterministic choices during the tableaux expansion.

The AKT portal ontology is also medium-sized. It contains 173 atomic concepts, 142 roles and 75 individuals, with 15 nominals (all in enumerations). The descriptions are not as complex as those in Wine and nominals are used, though

not heavily. Due to the presence of enumerations, nominal absorption reduces classification time. Lazy graph generation, graph caching and learning-based disjunct selection also have an influence in the results.

The 3SAT ontology uses nominals for encoding the 3SAT problem in OWL-DL. Due to the way the problem has been encoded, the ontology contains just 1 atomic concept, no roles and 20 nominals. For this case, nominal absorption and graph caching are especially effective. Both techniques speed up consistency checking time in three orders of magnitude.

Finally, we have run an experiment with a modified version of the Wine Ontology, containing *pseudo-nominals*. Since traditionally DL reasoners do not support reasoning with nominals, the *pseudo-nominal* approach tries to approximate the enumerated class definitions by replacing each nominal $\{o\}$ with a fresh atomic concept P_o and adding the assertion $P_o(o)$ to the ABox. Reasoners such as Racer and KAON2 adopt this technique and are not complete w.r.t. nominals.

We have run 10 independent experiments with all the optimizations enabled to classify and realize the modified Wine ontology containing *pseudo-nominals*. We have obtained the following results: 541ms for consistency, 2423ms for classification and 158648ms for realization. Note that, since the ABox does not influence reasoning in the TBox, due to the absence of nominals, consistency and classification times are faster; however, a high computational price is paid in realization since nominal-based model merging cannot be used any more. Overall, the total processing time is 1 order of magnitude *slower* with pseudo-nominals. This result indicates that *faking* nominals can be more costly, especially when nominals are used heavily in

the ontology.

Very recently a new version of FaCT++ reasoner supporting nominals was released. FaCT++ version 1.0.0 supports the DL $\mathcal{SHOIQ}$(D). However, this version of FaCT++ does not support ordinary ABox assertions[1] so it was not possible to run some of the above experiments or measure consistency checking and realization times separately. For this reason, we have only tried one experiment: classifying Wine ontology using FaCT++ 1.0.0. We have used a timeout of 30 minutes and classification was not completed in any experiment in the allowed time frame. This result also supports our hypothesis that without specific optimizations, reasoning with nominals is not practical.

We can summarize our results as follows:

1. It is not practical to reason with nominals without having special optimizations, especially when the ontology uses nominals heavily.

2. Nominal absorption has proven the most useful technique and has a significant impact, even in presence of a marginal number of nominals in the ontology.

3. Learning-based disjunct selection is particularly effective in the presence of individuals with similar characteristics, as shown in the OWL-S case.

4. Nominal-based pseudo-model merging is only useful on ontologies with *hasValue* restrictions [2] and affects primarily classification and realization times.

[1]Theoretically, ABox individuals can be encoded as nominals and ABox assertions can be turned into inclusion axioms but such an automated transformation was not available

[2]Wine is the only ontology in our experiments that contains hasValue restrictions

5. Lazy graph generation and graph caching can have a dramatic influence on concept satisfiability and subsumption tests.

6. The pseudo-nominal approximation is not only unsound, but may actually degrade the reasoner's performance.

Evaluating Conjunctive Query Answering Optimizations

In our experiments, we first tested the accuracy of the size estimation. For this purpose, have used some of the existing benchmark problems for conjunctive queries: the data from Lehigh University Benchmark (LUBM) [125] and ontologies Vicodi and Semintec from [89]. The following tables show the number of individuals in each dataset, the time spent for estimating the size of all the concepts in each dataset, and the mean normalized error over all concepts in the given ontology. For example, an error of 3.6 means that if the actual number of instances for a concept was 200, the algorithm returned 200 ± 7.2. We have varied the sampling percentage from %20 to %100.

As expected, error in size estimation decreases as we inspect more and more individuals. More interestingly, for these ontologies, sizes can be computed with ei-

		Sampling Percentage				
Dataset	Size	%20	%40	%60	%80	%100
LUBM	55664	3.6	6.7	9.7	11.8	15.0
Semintec	17941	0.9	1.6	2.4	3.2	3.9
Vicodi	16942	1.8	3.4	5.1	6.9	8.6

(a) Time spent in seconds

	Sampling Percentage				
Dataset	%20	%40	%60	%80	%100
LUBM	0.7	0.3	0.6	0.4	0.0
Semintec	6.4	4.4	4.9	3.7	0.0
Vicodi	18.5	11.3	7.6	4.7	0.9

(b) Mean normalized error

Table 7.2: Evaluating size estimation performance and accuracy with respect to sampling ratio.

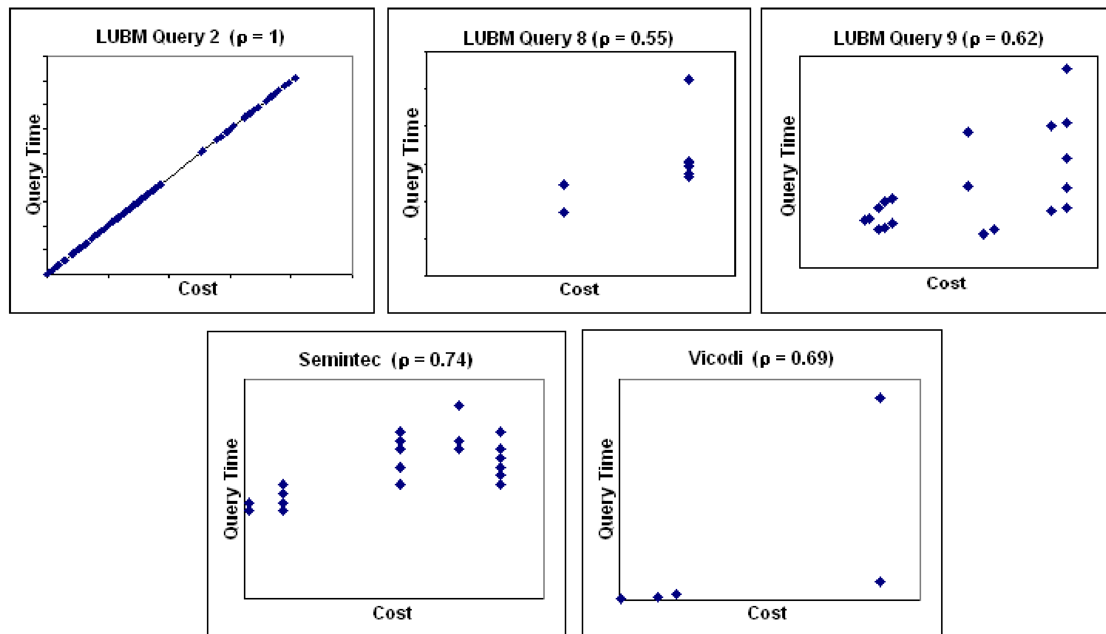

Figure 7.6: The correlation between the cost estimates and the actual query evaluation time. Each data point represents a different ordering of the corresponding query. Note that, due to simplification and heuristic pruning, number of data points is less than all possible orderings.

ther perfect or nearly perfect accuracy if all the individuals are inspected. However, computation time also increases. Looking at these results we decided to use a sampling ratio of %20 which yields fairly accurate results with reasonable computation time.

Next, we looked at the effectiveness of the cost model defined in this paper. Query ordering has a significant effect especially when there are many atoms in the query. Therefore, we used three conjunctive queries, Q2 (6 atoms), Q8 (5 atoms), Q9 (6 atoms), from LUBM and Q2 (5 atoms) from Semintec and Q2 (3 atoms) from Vicodi. We generated all the possible query orderings, pruned the orderings based on the aforementioned heuristics and computed the time to answer each query with that ordering. Figure 7.6 shows the scatter plot of different query orderings

	LUBM Q2	LUBM Q8	LUBM Q9	Semintec	Vicodi
Min Cost Ordering	20	320	227	100	81
Min Time Ordering	10	285	164	100	81
Median	1183	341	311	135	255
Max Time Ordering	1233	348	400	140	2123

Table 7.3: Comparison of query evaluation times for different orderings

where X axis is the estimated cost and Y axis is the actual time it took to generate the answers. The correlation factor for each query is different ranging from low (~ 0.5) to perfect score ($= 1$). More importantly, in each case, the lowest-cost query ordering found by the estimation algorithm is very close to the optimal value.

Table 7.3 shows these results in more detail and displays query evaluation time (in milliseconds) for the minimum cost ordering, the minimum time ordering, the maximum time ordering and the median of all orderings (still excluding the heuristically pruned orderings). Although the minimum cost ordering does not always take minimum time, we can still see that the improvement in query evaluation time compared to an arbitrary ordering can be more than one order of magnitude.

7.5.2 Planning Performance

We wanted to investigate two different points in our experiments: 1) How does HTN-DL perform for solving benchmark problems designed for classical planners 2) How do the new task matching/ranking mechanism scale to a large number of services and complex ontologies.

The objective of the first experiment is to see the overhead of using a DL reasoner for evaluating preconditions and effects of the services compared to the

original planning system where the state of the world is represented simply as a relational database. In order to have a planning problem that would run on both systems, we took a standard planning problem (Rover domain) which was used in the 2002 International Planning Competition [32]). In this domain, a collection of rovers navigate a planet surface, finding samples of rocks, analyzing the samples, and communicating the results back to a lander. We encoded this domain in HTN-DL using a DL ontology. This encoding changes the original planning problem in several ways:

1. Some facts about the state, such as the type of rovers involved and the rocks on the planet, are described using the ontology. As a consequence, evaluating the conditions of operators and methods resort to theorem proving rather than simple database queries.

2. This domain, as many other standard planning domains, uses n-ary predicates to represent some relations, e.g. $can_traverse(rover, loc1, loc2)$. We translated such relations using extra individuals, e.g. ($can_traverse(rover, path)$, $begins(path, loc1)$, $ends(path, loc2)$. For this reason, HTN-DL versions of problems were larger compared to the original problems, especially the harder problems contained significantly more number of individuals.

3. By following the example of [30], we "opened-up" the initial state by making the knowledge about some of the predicates, such as the location of a rover, incomplete.

Although the HTN-DL translations of the problems were considerably different

than the original problems to get an idea about the effectiveness of HTN-DL system, we compared the performance results to JSHOP, the Java version of the SHOP planner [95]. This is not really a fair comparison as the input planning problem to the planners is different. The reason for using JSHOP (instead of another planner that can better represent incomplete information) is because the most important factor that affects planning time is the domain knowledge encoded in the planning domain as seen in the results of the International Planning Competition [32]. Using a different system that has a different way of representing control knowledge would be even harder.

There were 20 different problems of increasing complexity provided in the planning competition and we ran both systems 20 times on each problem and computed the average time spent on planning. Figure 7.7 shows (in logarithmic scale) the total planning time spent by both systems for the different problems in the suite. As seen in the figure, the total planning time for HTN-DL is generally slightly greater than JSHOP. However, for the hardest problem, where there are large number of resources, HTN-DL performance is better than JSHOP. This is due to the fact that DL reasoner Pellet used in HTN-DL is optimized to handle large number of instances whereas the JSHOP implementation is not. This experiment shows that even though the expressivity of the knowledge representation language increases dramatically the reasoning time does not necessarily increase.

To test the performance of task matching/ranking in the presence of large ontologies, we have created a planning domain about obtaining books using Web Services. We created different versions of book-obtaining services which have dif-

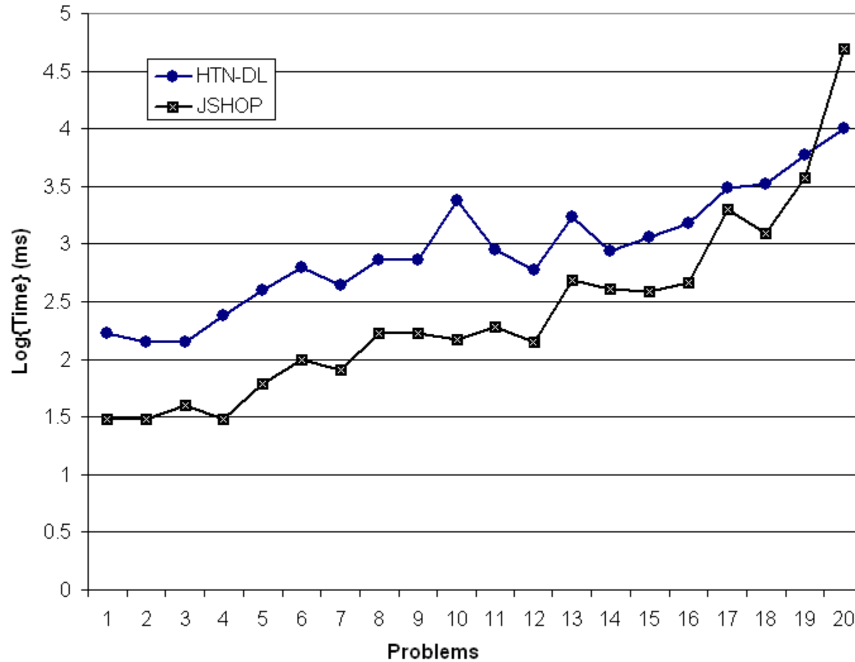

Figure 7.7: Comparison of HTN-DL with JSHOP system

ferent conditions, e.g. some online stores require registration before you place an order, a university library on the other hand will lend books only to its students and faculty. The planner needs to find a service for each book in the request list, verify the availability, ensure all the conditions of the service are met. The planning problem is to buy one or more books where there are different restrictions for each book, e.g. for a certain book we may be interested in only unused copies sold by a high rated service.

Note that this is a relatively simple problem from a planning perspective but has quite different characteristics than a usual planning problem. Classical planning benchmark problems, e.g. famous blocks world problem, has generally dealt with a small number of predefined actions, e.g. move block, in the presence of large number of objects, e.g. hundreds of blocks. However, interesting Web Composition

problems generally deal with a large number of actions with varying properties, e.g. hundreds of book selling services, but with limited number of objects involved, e.g. buying a couple of books. For this reason, we believe this setting is a good starting example that shows the characteristics of a Web Services domain.

For describing the task ontology, we have augmented the OWL version of the North American Industry Classification System (NAICS) ontology with some additional definitions. NAICS ontology contains definitions about 1800 categories for classifying business establishments. We have used the categories such as "Book stores" (NAICS code 451211), "Used Merchandise Stores" (NAICS code 453310), "Electronic Shopping" (NAICS code 454111) and specialized these classes for book buying services .

Figure 7.8 shows the planning time for solving different versions of this problem. We have randomly generated planning domains with 50, 100, 250, 500 and 1000 services and planning problems that involved buying 1, 2, 3, 5, and 10 books. For each setting, we used 10 different problems and reported the average planning time. Note that buying 10 books using 1000 services takes only 1.4sec which is quite reasonable. Optimized instance retrieval algorithm used in task matching show a linear behavior even for the cases where we have 1000 instances (i.e. services) and 2000 concepts (i.e. categories) in the task ontology.

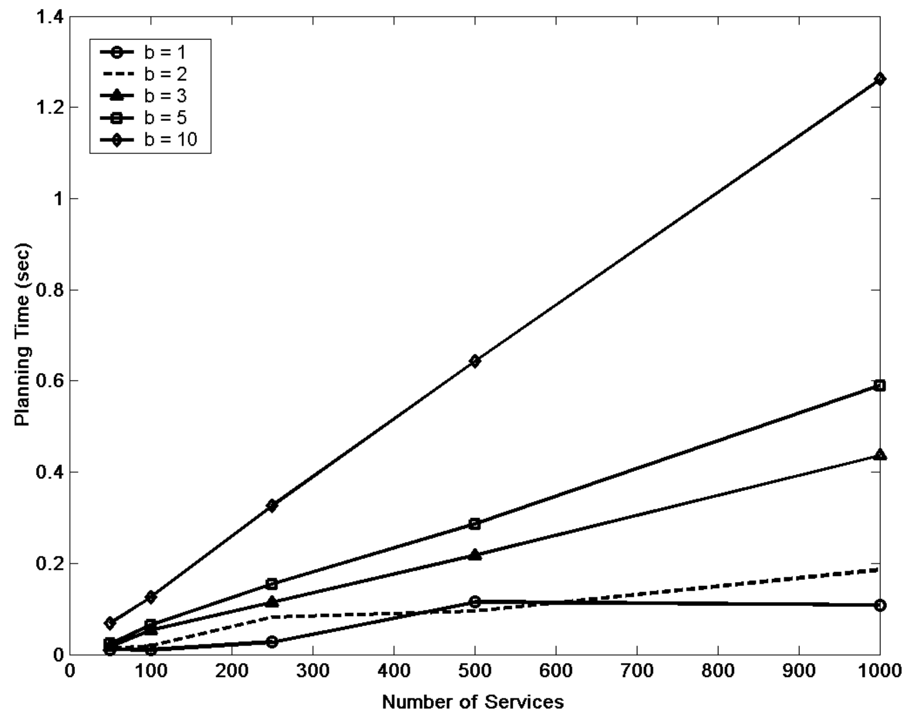

Figure 7.8: Performance of HTN-DL planner where each line shows the time spent with a different size of input task network

Chapter 8

Related Work

8.1 Description Logics and Planning

There are several different ways that Description Logics and planning systems have been combined in the past. DLs have been used to represent the state of the world, the actions used in planning and the plans generated. We will review the different types of combinations and examine how they relate to HTN-DL.

The most common approach of using DL to represent the state of the world is to use a propositional representation of planning problems where DL concept descriptions represent different states. CLASP (CLassification of Scenarios and Plans) [24] is one such system that uses the Classic [14] Description Logic to represent states and actions. The actions are described in the STRIPS style with preconditions, add lists and delete lists. These expressions as well as states can be expressed either as a primitive DL concept or a conjunction of primitive concepts. A specific state is represented as an instance of conjunctive concept. An action is applicable if the current state is an instance of its precondition expression. State transition is computed by removing/adding elements from/to the conjunctive concept as defined by the action's add and delete lists. In this regard, the planner uses Closed World Assumption and requires complete knowledge about the problem to be given.

This approach to represent state is very closely related to that of De Giacomo

et. al. [20, 21, 68] where the tight correspondence between Propositional Dynamic Logics (PDL) [110, 22] and Description Logics is used to reason about actions. The correspondence is realized through a (onetoone and onto) mapping from PDLs formulas to DLs concepts, and from PDLs actions to DLs roles. The state constraints and effect axioms are described as inclusion axioms. The planning problem can be posed as proving the subsumption $Initial \sqsubseteq \exists Any^{*}.Final$ where $Initial$ is the DL concept describing initial state, $final$ is the DL concept describing the goal state, Any is a universal super role, and the operator * is the transitive closure operator. Subsumption testing would be reduced to unsatisfiability and the plan would be reconstructed from the proof of inconsistency. This representation can encode concurrent actions (using role conjunction) and nondeterminism between actions (using role disjunction) but has the disadvantage of requiring complete knowledge about the world as pointed out by [7]. To overcome this problem a DL that incorporates non-monotonic operators, namely *minimal knowledge operator* **K** and *default assumption operator* **A**, can be used [21, 68]. However, since no reasoner supporting such an expressivity exists, the implementation of such a system resorts to procedural rules of Classic making it somewhat limited.

In a similar vein, Badea [7] proposes a satisfiability based encoding of planning problems in the spirit of [72]. This time the plan is constructed from a model constructed by the DL reasoner. However, the complete knowledge requirement still exists.

RAT (Representation of Actions using Terminological logics) [50] also uses concept expressions to describe state of the world. But different from the previous

approaches, it has a more expressive precondition language where agreements and disagreements between functional roles can be used to describe the relations between action parameters. An action can be applied to a state if the precondition expression subsumes the current world state.

Representing action hierarchies is another area where DLs have been used. The main purpose is to provide an operation that can decide subsumption between actions similar to the subsumption between concepts. CLASP system defines two different kinds of subsumption between actions: "psubsumption" and "gsubsumption". Kemke [74, 73] provides a slightly different subsumption definition. Liebig and Rosner [79] categorizes different subsumption techniques as follows:

- **AbstractionSubsumption:** Abstractionsubsumption means that an action A1 subsumes another action A2 iff A1's pre and postconditions are more general than those of A2. Kemke's definition conforms to this kind of subsumption, as well as the "psubsumption" in CLASP.

- **GoalSubsumption:** This relation organizes actions simply by considering the subsumption ordering of the goal description. This is useful, if a library is searched for an action which secures the achievement of a given goal. This is called "gsubsumption" in CLASP.

- **ApplicabilitySubsumption:** According to applicabilitysubsumption, an action A1 subsumes an action A2 iff A1 is always applicable when A2 is. In this case, it is necessary that A1 has a weaker precondition and a stronger postcondition than A2.

Liebig and Rosner argues that different type of action subsumption is appropriate in different settings. Although the precondition and effect languages in these systems are quite different compared to HTN-DL, one can see the conceptual similarity between the task matching in HTN-DLand the definition of applicabilitysubsumption. Since in HTN-DL, we are interested in replacing the task with an executable concrete action, it is natural that applicabilitysubsumption is fitting for our purposes.

Action subsumption is closely related to *plan subsumption* and *plan recognition*. CLASP system provides an extension to CLASSIC language where plans can be defined from actions using operators SEQUENCE, LOOP, REPEAT, TEST (conditional branching), OR (disjunctive branching), and SUBPLAN. The SUBPLAN construct supports modular definitions of plans through definitions of meaningful sub-networks. Plan subsumption is then defined based on p-subsumption and g-subsumption notions we explained earlier plus the subsumption between plan expressions that are described using the above constructs. There is no recursion allowed in the plan expressions so it is possible to represent each plan expression as a regular expression. Then plan subsumption can be done by checking if the language accepted by one deterministic finite automata (DFA) is subsumed by the other DFA. Using this approach, CLASP can classify plans into a plan taxonomy and determine if a specific plan instance, called a *scenario*, satisfies a plan expression. Although this approach is quite useful since HTN-DL is more expressive regular expressions (even totally ordered task networks can encode context-free languages [37]) it is not directly applicable to HTN-DL.

The work of Baader et. al. [4], although not directly dealing with planning,

is closely related to HTN-DL. In their paper, authors propose to describe Web Services using DL-based preconditions and effects similar to HTN-DL. Although their service descriptions are ground, i.e. there are no input variables, their approach can be easily extended to services with parameters. Authors provide a model-based update semantics for their service descriptions. There are two problems considered regarding reasoning about actions: *executability* and *projection*. Executability is the problem of deciding if the preconditions of the service is satisfied in the current state. Projection is to check if a certain condition holds after the execution of several services. It is shown that executability and projection can be reduced to each other and complexity results for a variety of DLs is provided. The limitations of the service descriptions, e.g. restriction to only acyclic TBoxes, not using transitive properties, allowing only primitive concepts in effects, are justified to avoid semantic problems that are similar to those for disjunctive postconditions. The semantics provided for the updates in this work is complementary to HTN-DL and as we discussed in Section 3.2.3 and 3.3.1 could be integrated to HTN-DL.

8.2 Description Logics and Web Services

In this section, we will review the related work in the area of DLs and Web Services. In particular, we will present the existing work on how DL reasoning has been used to facilitate Web Service matchmaking. We will discuss how the task matching features of HTN-DL relate to these approaches.

Research on matching Semantic Web Services has primarily focused on using

the subsumption relation between Web Service advertisements and requests. And more specifically the subsumption relation between the input and output types have been used to generate matchings for Web Services that were defined using OWL-S. The OWL-S Matchmaker [97] is the first system that implemented this idea in a system.

The Matchmaker system uses OWL-S profiles to describe service requests as well as the services advertised. A service provider publishes a DAML-S description to a common service repository. When someone needs to locate a service to perform a specific task, a ServiceProfile for the desired service is created. Request profiles are matched by the service registry to advertised profiles using DL subsumption as the core inference service. In particular, the Matchmaker computes subsumption relations between each individual input, output, precondition and effect (IOPE) of the request and the advertisement Service Profile. If the classes of the corresponding parameters are equivalent, there is an exact and thus best match. If there is no subsumption relation, then there is no match. Given a classification of the types describing the IOPEs, the Matchmaker assigns a rating depending on the number of intervening named classes between the request and advertisement parameters. Finally, the ratings for all of the IOPEs are combined to produce an overall rating of the match. In summary, the basic rating used in matchmaking are as follows:

- **Exact** If advertisement A and request R are equivalent concepts, it is called an Exact match
- **PlugIn** If request R is sub-concept of advertisement A, it is called a PlugIn

match

- **Subsume** If request R is super-concept of advertisement A, it is called a Subsume match

- **Fail** Otherwise, there is no match

There have been several proposals [43, 78] to extend the matchmaking algorithms to exploit more features of subsumption relations. For example, when there is no subsumption relation between the advertisement and request, a rating called *Intersection* may be assigned when their intersection is not empty, i.e. advertisement and request descriptions are not disjoint. This case implies that relaxing some of the constraints on the request may provide better results. And both approaches differ from the Matchmaker because they use the whole service description, or more correctly the profile description, for discovery purposes and try to find the subsumption relation between these more complex class expressions. Li and Horrocks [78] point out a problem about OWL-S profile descriptions where encoding too much information in the profile, e.g. name and address of the provider, prevents effective matching. They overcome this problem by separating various components of the description; in particular the description of the service being provided was separated from the descriptions of the providing and requesting "actors".

The main difference of task matching in HTN-DL is that we consider precondition and effect expressions in the matching criteria. Without any precondition and effect descriptions, our task matching criteria would indeed boil down to pure type-based matching. But as we pointed out in Section 3.2.1, type-based matching

is not enough to differentiate functionality of parameters that has same types.

The METEOR-S Web Service Composition Framework (MWSCF) [111] is also relevant as it allows users to describe abstract Web Service workflows. MWSCF describes Semantic Process Templates (SPT) that describe a workflow of abstract and concrete services. The templates may include QoS criteria that will be used for discovery. For ranking discovered services, each selection criteria is assigned a numerical score and a weighted combination of these scores are computed to rank the services. The main difference of our approach compared to MWSCF is the way we make use of non-deterministic choice constructs to encode possible different execution paths and consider recursive decomposition of templates. MSWCF on the other hand focuses on matching abstract services with atomic concrete services.

8.3 Web Service Composition and Planning

Several different AI planning techniques have been proposed to automate Web Service composition. Most of these systems are based on causal planning where there are only primitive actions but no composite actions. The state representation is also limited to set of ground atoms and do not consider domain axioms. In what follows, we will describe these planning systems in more detail and discuss how they differ from HTN-DL.

The first approach for composing OWL-S services was described in [88] and is based on the notion of generic procedures. This work extends the Golog language to enable programs that are generic, customizable and usable in the context of the

Web. However, in Golog such programs are defined as macros and they are complied away. So it is not possible to describe non-functional attributes of such programs or use these attributes for flexible matching as in HTN-DL. The composition system uses an augmented Golog interpreter that combines online execution of sensing actions with offline simulation of world altering services. HTN-DL is very similar to this approach in spirit. However, as discussed in Section 4.3, the Invocation and Reasonable Persistence (IRP) assumption of [88] prevents the planner to change (i.e. simulate the changes) the information gathered from external sources. One advantage of using situation calculus as the underlying logical framework is the additional expressivity and the ability to do arbitrary reasoning about first-order theories. However, Golog implementation uses regression to reason about actions, i.e. to solve executability and projection problems. As discussed in detail in [4], translating OWL-S descriptions (or descriptions of similar expressitivity) to situation calculus and applying regression yields a standard first-order theory which is not in the scope of what Golog can handle without calling a general first-order theorem prover.

In [87] a technique based on *estimated-regression* planning is proposed for generating compositions of Web Services. The estimated-regression planner Optop is used for this purpose. In Optop, a state of the planner is a *situation*, which is essentially the current partial plan. Optop works with classical-planning goals; thus, it checks whether the current situation satisfies the conjunction of the goal literals given to the planner as input. During its search, Optop computes a *regression-match graph* as described in [87], which essentially provides information about how

to reach to a goal state from the current situation. The planner returns the successor situations that arises from applying the actions specified by that graph in the current situation.

In a different approach, [82] proposed to model the services and the information about the world by using the "knowledge-level formulation" first introduced in the PKS planning system [103]. This formulation involves modeling Web Services based on not what is actually true or false about them, but what the agent that performs the composition actually knows to be true or false about their operations and the results of those operations. In this approach, a composition is formulated as a conditional plan, which allows for interleaving the executions of information-providing and world-altering services. HTN-DLplans are not conditional because our approach is based on executing the information-providing services during planning to clear out the "unknown"s during planning time as much as possible. Modifying HTN-DL algorithm to generate conditional plans would be valuable in order to handle changing information. The other important difference of HTN-DL is the treatment of knowledge. We do not model preconditions based on the knowledge of the planning agent.

Another approach for Web Service composition is proposed in [104] and [113]. This approach is a planning technique based on the "Planning as Model Checking" paradigm for the automated composition of web services described in OWL-S process models. The OWL-S process models are translated into state transition systems that describe the dynamic interactions with external services. The composition goals are expressed in a language where temporal restrictions on goals and preferences about

goals can also be specified. With the composition goal and the state transition systems, the planner, based on symbolic model checking techniques, returns an executable process rather than a linear sequence of actions.

The XII [41] planner and its successor PUCCINI [40] was developed well before Web Services came into existence but both planners designed to interact with so-called *softbots* (SOFTware roBOTs). The SADL language developed for PUCCINI distinguishes between (1) the world-altering and the observational effects of the actions, and (2) the goals of satisfaction and the goals of information. The planner can work under incomplete information but is also able to represent closed world features using Local Closed World (LCW) representation [28]. LCW allows the planner to know that it has complete knowledge about a subset of the state of the world. PUCCINI algorithm, which is an extension of UCPOP [101], interleaves execution and planning by invoking (mostly sensing) actions during planning. If executed actions have some side-effects and the planner needs to backtrack, the *reversing actions* are used to undo the side-effects. Although the state of the world representation in PUCCINI is limited to ground set of literals but truth value of literals are three-valued as in HTN-DL. There is also a strong similarity between the LCW axioms and nominals in DLs as nominals can be used to "close the world" by enumerations. Certain LCW axioms can be seen as syntactic variations of DL axioms using nominals [49].

Chapter 9

Conclusions

9.1 Summary

In this thesis we have identified some of the challenges of automated Web Service composition problems; namely, *decentralized setting*, *handling service attributes*, *composite services*, *reasoning with open world semantics*, *need for interleaved execution and composition* and *efficiency*.

To address all these challenges, we presented HTN-DL which combines the HTN planning formalism with DL representation. There are many novel features of HTN-DL that make it suitable for solving Web Service composition problems. The service categorization and non-functional attributes of services are described in a task ontology that allows flexible matchmaking. The state of the world is also represented as a DL knowledge base so that we have a much more expressive language with Open World semantics. HTN-DL also differentiates between world-altering effects and knowledge effects making it possible to interleave planning with execution by invoking information-providing services during composition.

We show that HTN-DL methods are capable of representing control constructs that are commonly used to model composite Web Services. Specifically, we provided an algorithm to translate process models expressed in the Semantic Web Service language OWL-S to HTN-DL. This translation provides a semantics for OWL-S process

models. We also show this semantics is compatible with a previous proposal that gives a semantic for OWL-S in terms of situation calculus.

As the planning system relies on the inferences drawn by the DL reasoner, the practicality of the proposed solution crucially depends on the efficiency of the DL reasoner. For this reason, several novel optimization techniques, especially geared toward handling nominals and large number of individuals, are presented in this thesis. These are the first optimization techniques developed for nominals and empirical analysis shows that these techniques can drastically improve the performance of consistency checking, classification, and realization tasks. The other frequently used reasoning service by the the HTN-DL planning system is conjunctive query answering. To improve query evaluation times, optimization techniques for conjunctive query answering inspired by the techniques used in relational databases are presented.

Finally we presented our prototype implementation for HTN-DL. We described the system architecture of the HTN-DL components; namely, Pellet, the OWL-DL reasoner, and OWL-S API, an API for managing and executing Web Services. We also present the empirical evaluation of the system performance.

9.2 Contributions and Impact

The contributions of this thesis are as follows:

- HTN-DL formalism, which couples HTN planning and Description Logics, combines the expressivity of Description Logics with the efficiency of HTN plan-

ning systems to solve Web Service composition problems. The hierarchical structure of HTN-DL domains can conveniently describe composite Web Service descriptions and fit in well with the loosely coupled nature of Web Services. Ontology-based reasoning provides a flexible mechanism to reuse the Web Services that are defined by separate developers in different contexts.

- A translation algorithm from OWL-S to HTN-DL is provided showing that the control constructs used to describe the control flow of a Web Service workflow can be encoded as HTN-DL domains. The translation provides a semantics for OWL-S processes and is shown to be compatible with the previously proposed Situation Calculus based semantics of OWL-S.

- Novel optimizations for DL reasoning targeting nominals and large number of individuals are presented. Our empirical analysis shows that these optimizations drastically improve consistency checking, classification and realization tasks.

- Optimizations for conjunctive query answering w.r.t. DL knowledge bases are introduced. Inspired by query optimization techniques used in relational databases, a cost-based model is presented to estimate the evaluation time of DL queries. We propose efficient heuristics to compute the costs of queries and demonstrate the effectiveness of the query optimization techniques empirically.

- An implementation of HTN-DL planning system that interacts directly with Web Services is presented. The components of the planning system, OWL-DL

reasoner Pellet and API for OWL-S services, are also released as stand-alone tools and have been incorporated in many systems.

HTN-DL is the first formalism to combine planning with expressive knowledge representation. It has an efficient implementation and it provides a promising solution for Web Service composition problems. The optimizations for DL reasoning presented in this thesis are not only applicable in Web Service composition problems but improve DL reasoning performance for classification, realization and conjunctive query answering. For example, Pellet can classify the notoriously hard Wine ontology using the optimizations described in this thesis. The DL reasoner Pellet and OWL-S API are also released as stand-alone open-source software. They are incorporated in several academic and industrial projects such as *Task Computing Environment* of Fujitsu Labs and *Swoop* in University of Maryland.

9.3 Discussion

The HTN-DL formalism tackles many of the problems associated with automating the composition of Web Services but there are still many issues that need to be addressed. HTN-DL can serve as basis for a more extensive Web Service composition framework that deals with these issues.

One problem we have not addressed throughout the thesis is the process of *discovering* Web Service descriptions. The HTN-DL planner assumed that all the service descriptions would be provided as input to the system. This is not very realistic as we cannot expect the user's planning agent to crawl the Web to find all

available Web Services. There are obviously practical issues because the number of available Web Services is most likely to be very high. The anticipated way of discovering Web Services is through Web Service registries where providers publish the descriptions of their Web Services and users search the registry for an appropriate service. Universal Description Discovery and Integration (UDDI) [114] is a standard proposed for such a Web Service registry system based on WSDL descriptions. Paolucci et. al. [98] discusses how to extend the UDDI architecture to enable matching based on Semantic Web Services descriptions. A similar extension can be designed for HTN-DL.

We also did not discuss in this thesis how one can trust the Web Service description will actually perform the task it claims to perform. The service description might say that it sells books, but we do not have a guarantee that the service will ship the book after the credit card is charged. Similar situations exist when a human user is manually finding and executing services on the Web. However, automating the generation of composition process makes the problem more serious because the control and involvement of a human (who is supposed to make the decision whether to use the service or not) is now minimized. We discussed in Chapter 3 how task ontologies might be used to describe non-functional attributes of a service that might include policy descriptions. The task matching might be limited to services that satisfy certain policy requirements. Although there are several proposals — including WS-Policy [122] and WS-Trust [123] — to describe such properties, the standards are still very far away from allowing automatic verification of Web Service descriptions.

One other problem we overlooked in this thesis is changing information. It is possible that, during planning, some of the information the planner has will be changed by external agents. We have no control over such events. It might not always be possible to detect such cases. For example, we might gather the information that a book is in stock in a bookstore, but before we execute the generated plan, the last copy of the book might be sold to someone else.

9.4 Future Work

There are several different research directions which can improve and extend the work presented in this thesis:

- **Global Consistency** In this work we made the global consistency assumption, which requires the state of the world to be consistent and the updates to yield consistent world states. However, this consistency requirement is not achievable in a distributed environment such as the Web. There are at least two different types of inconsistency that can arise. First, there can be a contradiction between domain ontologies used to describe services. Second, even if the domain ontologies are consistent, the information we gather from remote services might be contradictory with each other or with the domain ontologies. For example, the first kind of contradiction arises if two service ontologies describe the same concept in a conflicting way, e.g. one stating the a flight itinerary can be associated with at most one flight where the other ontology describes itineraries with multiple flights. The second type of inconsistency

can occur if one service says there is a seat available in one specific flight where the second service says that flight is completely booked.

In [75], we provided a simple and effective, yet not a completely satisfactory, solution for these kinds of inconsistencies by using trust metrics associated with information-providing Web Services. If each service is associated with such a ranking, we can start the execution of services with the highest ranked service and ignore the results of a service if it contradicts with previous information we collected. This simple sort of ranking scheme seems well suited for Web Service developers and easy to integrate with WSDL or OWL-S descriptions. However, this "all-or-nothing" approach has the disadvantage of discarding parts of the information that might not be contradictory. Applying this similar strategy to domain ontologies would be even harder because a contradiction does not mean imperfect or faulty information as in the previous case but indicates a more intrinsic problem in the way these ontologies model the world. In such cases, using a paraconsistent logic that is robust in the face of inconsistencies would be more appropriate. The kind of paraconsistency required and how this will affect reasoning procedures is subject to future research.

- **Web Service Choreography and Multi-agent Aspects** One of the motivations behind HTN-DL was to deal with Web Service descriptions developed by separate providers. Although these services are decentralized, they are all considered to be *reactive*; that is, the composition system finds, executes, and handles coordination between them. In that sense, the HTN-DL framework

is motivated toward *orchestration* of Web Services. On the other hand, it is possible to have more *proactive* services that can act autonomously. The *choreography* of such services is closer to multi-agent systems where multiple agents share or compete with resources and knowledge to solve planning problems. In such settings some of the remote services could be capable of solving a subproblem of our overall goal, e.g. achieve one of the tasks in the input task network, and we can delegate this problem to another agent.

- **Preferences and Optimal Compositions** This thesis concentrates on representing Web Service composition problems and efficiently finding compositions. But we do not address the preferences of the users regarding the services used or the plans generated. User preferences can be specified in the composite service descriptions to control how task matching will be done, e.g. rank the possible matching services and use the most desire one. Or users might specify some overall optimality conditions such as minimizing the total cost of composition. However, ensuring that the plan found is optimal with respect to such conditions would typically not be practical in the Web Services context. Since there are many different services that can be used for each task, exploring all possibilities cannot be achieved in a reasonable time. We hypothesize that an any-time algorithm that is capable of generating near-optimal solutions would be appropriate in this setting.

- **Planning without Method Descriptions** The HTN-DL planner, as any other HTNplanning system, requires the existence of method definitions for

decomposing a compound task into smaller tasks. In the event that there are no matching tasks, the planner fails to find a plan. One way to overcome this problem is to use a mixed-initiative planning system where the user helps to find or build an appropriate method description for the missing step. Such a solution has been successfully integrated into existing HTN planning systems, e.g. see the HiCAP [1] system that employs SHOP2 [94] as a component of a mixed initiative system. Another possible solution is to learn HTN domains from existing plan traces as done in [67, 66, 34]. Looking at the previously executed plans, a system can automatically generate method descriptions. A similar approach has been adopted in the Web Services setting in the name of *workflow mining* (or process mining) [118, 116, 42]. The objective of workflow mining is to generate a Web Service workflow description from execution traces of Web Services. Given the correspondence between workflows and HTN methods we provided in Chapter 4, the combination of techniques developed in planning and workflow research might be fruitful for Web Services. And, finally, if there is not an available method to do task decomposition, one can use classical causal planning to fill in the missing step. Since the task descriptions in HTN-DL provide the expected effects of a task, a straight-forward causal planning problem would be described by using the effects of the task as the goal formula.

Appendix A

Proofs

Translation from OWL-S to HTN-DL

Theorem A.1 *Let $K = \{K_1, K_2, \ldots, K_m\}$ be a collection of OWL-S process models, C be a possibly composite process defined in K, S_0 be the initial state, and $P = (p_1, p_2, \ldots, p_n)$ be a sequence of atomic processes defined in K. Then P is a composition for C with respect to K in S_0 iff P is a plan for planning problem (S_0, T_C, D) where T_C is the task network containing the single task returned by the translation for process C, and D is the HTN-DL domain created from K.*

Proof The proof of the theorem is by induction:

Hypothesis For a given OWL-S process C, P is a plan for the planning problem (S_0, T_C, D) iff $\Sigma \models Do(\delta_C, S_0, do(\vec{a}, S_0)))$ where $\vec{a} = [a_1, a_2, \ldots]$ is the sequence of primitive actions in situation calculus that corresponds to the sequence of HTN-DL operators in P.

Base Case Suppose A is an atomic OWL-S process and a is the corresponding primitive action in situation calculus. Suppose o_A and t_A are the corresponding HTN-DL operator and task for A. Then in Golog it is defined that

$$Do(a, s, s') = Poss(a, s) \wedge s' = do(a, s)$$

It means when the preconditions for the process is satisfied with respect to situation s then the primitive action sequence we will get for this simple program will have only one element, namely $\vec{a} = [a]$. In line 5 of HTN-DL algorithm, `find-applicable` will return only o_A since that is the only operator (or method) in the domain that matches the task t_A when the preconditions of the operator are satisfied. As seen in line 10 the returned plan will contain only o_A since the recursive call will return empty list as there are no more tasks in the task network to perform. Thus, the plan returned by HTN-DL is $[o_A]$ which is equivalent to the situation calculus result.

Inductive Step We will do a case by case analysis for each of the control constructs in the process model to show that our translation and resulting plans HTN-DL finds are correct.

Choice Suppose C is a control construct defined as a `Choice` of two[1] other processes C_1 and C_2. The HTN-DL translation for C will yield a task t_C and two methods M_1 and M_2 such that both methods match t_C. Corresponding Golog program for C is $\delta_C = \delta_{C_1} \mid \delta_{C_2}$ and the semantics is defined as

$$Do(\delta_{C_1}|\delta_{C_2}, s, s') = Do(\delta_{C_1}, s, s') \vee Do(\delta_{C_2}, s, s')$$

The disjunction means any $\vec{a}$ that is a valid action sequence for either δ_{C_1} or δ_{C_2} will also be a valid sequence for δ_C. From our hypothesis we know for each action sequence $\vec{a}$ that satisfies δ_{C_1} (or δ_{C_2}) we have a valid HTN-DL plan P_{C_1} (or P_{C_2}).

[1] The Golog choice operator | is defined for two operands. A choice of more operands could be done by nested | operators which would not effect our proof here

The nondeterministic choices in HTN-DL algorithm (line 6 and line 12) shows that when a plan is being sought for t_C, any solution for any matching action instance, in this case M_1 and M_2, will be returned as a result. This is due to the fact that those two methods are the only ones in D that matches with task t_C. Furthermore there is only one simple reduction per method because both methods have only one task network containing only one subtask in them and the set of local variables,i.e. V, is empty. Thus the nondeterministic choice ensures that when HTN-DL is asked to find all the plans for C, both P_{C_1} and P_{C_2} will be returned proving the equivalence to the answer in situation calculus.

Sequence Suppose C is a control construct defined as a `Sequence` of two other processes C_1 and C_2. The corresponding Golog program for C is $\delta_C = \delta_{C_1}\ ;\ \delta_{C_2}$ and the semantics is defined as

$$Do(\delta_{C_1}; \delta_{C_2}, s, s') = (\exists s^*)(Do(\delta_{C_1}, s, s^*) \wedge Do(\delta_{C_2}, s^*, s'))$$

Suppose that situation s^* represents a history of the action sequence $\vec{a_1}$. If the action sequence recorded between situations s^* and s' is $\vec{a}_2$ then the final situation s' represents the concatenated sequence $\vec{a}$ =[$\vec{a}_1$, $\vec{a}_2$].

The HTN-DL translation for C_i will yield a pair of task and method t_{Ci} and M_{Ci} such that M_{Ci} matches M_{ti}. Similarly translation for C will return a task t_C and one method M_C such that M_C matches t_C. Furthermore M_C has the conditional task network $(\top : (\{u_1 = t_{C1}, u_2 = t_{C2}\}, \{(u_1, u_2)\}, \lambda))$. Calling HTN-DL($s$, t_{C1}, D) [2] will return P_{C_1} and from our hypothesis we know that it is equivalent to the action

[2] We are abusing the notation here by using the task t_{C1} instead of the task network containing only the task t_{C1}.

sequence $\vec{a_1}$. We also know that calling $\mathsf{HTN\text{-}DL}(s^*, t_{C_2}, D)$ will return a plan P_{C_2} that is equivalent to the action sequence $\vec{a_2}$. The HTN-DL algorithm shows that (line 14) when a task (in this case t_C) is removed from the input task network w, it is replaced with its simple reduction (in this case a task network containing t_{C1} and t_{C2} and additional edges preserving the order of the subtasks). The tasks to solve are selected from w in the order imposed by the edges in the network (line 4) so the resulting plan for $\mathsf{HTN\text{-}DL}(s, t_C, D)$ will actually be the concatenation of P_{C_1} and P_{C_2} which is equivalent to the sequence $\vec{a}$.

If-Then-Else Suppose C is a control construct defined as an `If-Then-Else` and *cond* is the condition for the if statement, C_1 is the process in the then part and C_2 is the process in the else part. Corresponding Golog program for C is δ_C = (**if** *cond* **then** δ_{C_1} **else** δ_{C_2} **endIf**) and the semantics is defined as

$$Do(\textbf{if } cond \textbf{ then } \delta_{C_1} \textbf{ else } \delta_{C_2} \textbf{ endIf}, \text{s}, \text{s'})$$

$$= Do((cond?; \delta_{C_1}), \text{s}, \text{s'}) \vee Do((\neg cond?; \delta_{C_2}), \text{s}, \text{s'})$$

$$= (cond[s] \wedge Do(\delta_{C_1}, \text{s}, \text{s'})) \vee (\neg cond[s] \wedge Do(\delta_{C_2}, \text{s}, \text{s'}))$$

The expression $cond[s]$ evaluates to true whenever the fluent *cond* is true in situation s. Suppose $\vec{a}_1$ is the action sequence for the situation δ_{C_1} and $\vec{a}_2$ is the action sequence for the situation δ_{C_2}. If s satisfies *cond* then the result for δ_C will be $\vec{a}_1$ otherwise result will be $\vec{a}_2$.

The HTN-DL translation for C_i will yield a pair of task and method t_{Ci} and M_{Ci} such that M_{Ci} matches M_{ti}. Similarly translation for C will return a task t_C and one method M_C such that M_C matches t_C. Furthermore M_C has the conditional task

network $(cond : (\{u_1 = t_{C1}\}, \emptyset, \lambda_1); (\top : (\{u_1 = t_{C2}\}, \emptyset, \lambda_2))$. From our hypothesis we know that for any possible $\vec{a}_1$ (or $\vec{a}_2$) we have a valid HTN-DL plan P_{C_1} (or P_{C_2}). When we call HTN-DL(s, t_C, D), the algorithm will check the conditions in the method definition (line 12), *cond* and $\top$ to find a simple reduction. If *cond* is satisfied then the algorithm returns P_{C_1} and otherwise returns P_{C_2} which is equivalent to the the result in situation calculus.

Repeat-While Suppose C is a control construct defined as a `Repeat-While` and *cond* is the condition for the while statement and C_1 is the process in the loop body. Corresponding Golog program for C is $\delta_C = (\textbf{while}\ cond\ \textbf{do}\ \delta_{C_1}\ \textbf{endWhile})$ and the semantics is defined as

$$Do(\textbf{while}\ cond\ \textbf{do}\ \delta_{C_1}\ \textbf{endWhile}, \text{s}, \text{s'}) = Do([[(cond?;\ \delta_{C_1})]^*;\ \neg cond?], \text{s}, \text{s'})$$

This definition includes the nondeterministic iteration operation `*` which has a second-order semantics [77]. We will use the restricted version of Golog as defined in [88] where the the iterations has a limit `k`. This restriction eliminates the problems caused by unlimited looping and enables us to define a first order semantics.

The HTN-DL translation for C_1 will yield a pair of task and method t_{C1} and M_{C1} such that M_{C1} matches M_{t1}. Similarly translation for C will return a task t_C and one method M_C such that M_C matches t_C. Furthermore M_C has the conditional task network $(cond : (\{u_1 = t_{C1} u_2 = t_C\}, \{(u_1, u_2)\}, \lambda_1); (\top : \emptyset, \emptyset, \lambda_2))$ where λ_1 and λ_2 are the parameter bindings handling the data flow.

Assume the iteration runs k times. When $k = 0$, the above formula will simplify to $Do(\neg cond?, \text{s}, \text{s'})$ which returns an empty action sequence in situation

calculus. This new formula also implies condition *cond* is false in the initial situation s. When HTN-DL is trying to solve M_C, since *cond* is false the algorithm will choose (line 12) the second condition-task list pair (note that the second condition in M_C is $\emptyset$ which is always true). The second task list is $\emptyset$ so HTN-DL will return an empty plan as well. Suppose $\vec{a}$ is a valid action sequence for δ_{C_1}. From our hypothesis we know for each action sequence $\vec{a}$ that satisfies δ_{C_1} we have a valid HTN-DL plan P_{C1}. In the general case, when $k > 0$, the Golog formula becomes $Do([cond?;\ (\delta_{C_1})^1;\ \ldots;\ cond?;\ (\delta_{C_1})^k;\ \neg cond?]$, s, s') hence the action sequence will be $[\vec{a}_1, \ldots, \vec{a}_k]$. Note that action sequence for each step of iteration may be different, for example when δ_{C_1} contains nondeterministic choices. We also know that *cond* will be true in situations $s, s_1, \ldots, s_{k-1}$ and false in situation s_k. When HTN-DL is searching a plan for t_C, the first condition (*cond*) will evaluate to true and HTN-DL will chose the first task network containing tasks (t_{C1}, t_C). Solving the first task t_{C1} will add P_1 to the plan and solving second task C will recursively continue until *cond* fails. Since, initial states are equal and plan prefixes are same, *cond* will not hold after kth iteration. At this point, algorithm will chose the second condition-task list pair (empty task list) which will conclude the recursion and the plan returned will be $[P_1, \ldots, P_k]$. At each step of the iteration we will have the equivalent world states so the action sequence a_i and plan P_i will be equivalent due to our hypothesis. Therefore, the final plan and the final action sequence will be equivalent.

Repeat-Until The proof for this case is very similar to the above proof for `Repeat-While` construct and is omitted. □

Nominal Absorption

Proposition A.1 (5.1) *The inclusion axiom (5.1) is logically equivalent to the set of axiom and assertions in (5.2)*

$$C \equiv \{a_1, \ldots, a_n\} \tag{A.1}$$

$$C \sqsubseteq \{a_1, \ldots, a_n\} \text{ and } C(a_1) \text{ and } \ldots \text{ and } C(a_n) \tag{A.2}$$

Proof The axiom (A.1) is equivalent to the combination of following two axioms

$$C \sqsubseteq \{a_1, \ldots, a_n\} \tag{A.3}$$

$$\{a_1, \ldots, a_n\} \sqsubseteq C \tag{A.4}$$

By the definition of enumerations, axiom (A.4) is equivalent to:

$$\{a_1\} \sqcup \ldots \sqcup \{a_n\} \sqsubseteq C \tag{A.5}$$

We can rewrite axiom (A.5) as the following n separate axioms:

$$\{a_1\} \sqsubseteq C \text{ and } \ldots \text{ and } \{a_n\} \sqsubseteq C \tag{A.6}$$

which is obviously valid based on the semantics

$$(\{a_1\} \sqcup \ldots \sqcup \{a_n\})^{\mathcal{I}} \subseteq C^{\mathcal{I}} \iff \{a_1\}^{\mathcal{I}} \subseteq C^{\mathcal{I}} \text{ and } \ldots \text{ and } \{a_n\}^{\mathcal{I}} \subseteq C^{\mathcal{I}}$$

Axiom (A.6) is equivalent to the following set of assertions:

$$C(a_1) \text{ and } \dots \text{ and } C(a_n) \tag{A.7}$$

because for each i we have

$$\{a_i\}^{\mathcal{I}} \subseteq C^{\mathcal{I}} \iff (a_i)^{\mathcal{I}} \in C^{\mathcal{I}}$$

since $\{a\}^{\mathcal{I}} = \{a^{\mathcal{I}}\}$.

Thus, we have shown that axiom (A.1) is transformed into the combination of (A.3) and (A.7) which is equivalent to (A.2). □

Proposition A.2 *The following two inclusion axioms are logically equivalent:*

$$\exists p.\{o\} \sqsubseteq C \tag{A.8}$$

$$\{o\} \sqsubseteq \forall p^{-}.C \tag{A.9}$$

Proof Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be a model of (A.8) s.t. it does not satisfy (A.9). Since $\mathcal{I}$ does not satisfy (A.9), then $o^{\mathcal{I}} \notin (\forall p^{-}.C)^{\mathcal{I}}$ which implies that $o^{\mathcal{I}} \in (\exists p^{-}.\neg C)^{\mathcal{I}}$. Thus, there exists an object $x \in \Delta^{\mathcal{I}}$ s.t. $(x, o^{\mathcal{I}}) \in p^{\mathcal{I}}$ and $x \in (\neg C)^{\mathcal{I}}$. On the other hand, since $\mathcal{I}$satisfies (A.8) and $x \in (\exists p.\{o\})^{\mathcal{I}}$, then $x \in C^{\mathcal{I}}$, which yields a contradiction.

Let $\mathcal{J} = (\Delta^{\mathcal{J}}, \cdot^{\mathcal{J}})$ be a model of (A.9) s.t. it does not satisfy (A.8). Since $\mathcal{J}$ does not satisfy (A.8), there exists an $x \in \Delta^{\mathcal{J}}$ s.t. $(x, o^{\mathcal{J}}) \in p^{\mathcal{J}}$ and $x \notin C^{\mathcal{J}}$. On

the other hand, since $\mathcal{J}$ satisfies (A.9), $o^{\mathcal{J}} \in (\forall p^{-}.C)^{\mathcal{J}}$ and, since $(o^{\mathcal{J}}, x) \in (p^{-})^{\mathcal{J}}$, then $x \in C^{\mathcal{J}}$, which again yields a contradiction. □

Nominal-Based Pseudo-Model Merging

Theorem A.2 *Let $\mathbf{G}' = (V', E', \mathcal{L}', \neq)$ be the initial completion graph for the concept C w.r.t the ontology $\mathcal{O}$ such that $V' = \{r_C, r_{o_1}, \ldots, r_{o_m}\}$ where r_C is the root node for concept C and r_{o_i} is the nominal node corresponding to nominal o_i. $\mathcal{L}'$ is initialized such that $\mathcal{L}(r_C) = \{C\}$ and $\mathcal{L}(r_{o_i}) = \{o_i\}$ for $1 \leq i \leq m$.*

Let $\mathcal{G}$ be the set of all complete and clash free graphs for C w.r.t. $\mathcal{O}$ that can be obtained from $\mathbf{G}'$ through the application of the expansion rules. If there is a role p s.t. for every $\mathbf{G} = (V, E, \mathcal{L}, \neq)$ in $\mathcal{G}$ there exists an edge $\langle r_C, r_o\rangle \in E$ with $p \in \mathcal{L}(\langle r_C, r_o\rangle)$, then, $\mathcal{O} \models C \sqsubseteq \exists p.\{o\}$.

Proof Let us assume that $\mathcal{O} \not\models C \sqsubseteq \exists p.\{o\}$. This means there should be an interpretation where there is an element that belongs to both concept C and $\forall p.\neg\{o\}$ (which is the negation normal form of $\neg(\exists p.\{o\})$. Then we should be able to build a clash free and complete completion graph starting with the initial graph $\mathbf{G}'' = (V'', E'', \mathcal{L}'', \neq)$ where $\mathcal{L}(r'') = \{C, \forall p.\neg\{o\}\}$. Since the graph $\mathbf{G}''$ is same as $\mathbf{G}'$ with one additional element in $\mathcal{L}(r)$, all the tableau rules applicable to $\mathbf{G}'$ will still be applicable to $\mathbf{G}''$. This means, every possible application of tableau expansion rules to $\mathbf{G}''$ will yield a member of $\mathcal{G}$ (with the additional element $\forall p.\neg\{o\}$ in $\mathcal{L}''(x)$). Then, by the assumption of the lemma, we know that $p \in \mathcal{L}''(\langle r, r_o\rangle)$ would hold. Therefore, the application of the $\forall$-rule would create a clash in $\mathbf{G}''$ since it would

add $\neg\{o\}$ to the label of r_o node. Hence we conclude no such clash free completion graph exists and $\mathcal{O} \models C \sqsubseteq \exists p.\{o\}$. $\square$

Lemma A.1 *Let $\mathcal{O} \models C \sqsubseteq \exists p.\{o\}$. Let $\mathcal{T} = (\mathbf{S}, \mathbf{L}, \mathbf{E})$ be a tableau for C w.r.t. $\mathcal{O}$. Then:*

1. *If p is a simple role, then, for any $s \in \mathbf{S}$ with $C \in \mathbf{L}(s)$ we have $\langle s, o\rangle \in \mathbf{E}(p)$*
2. *If p is not simple, there exists a role $q \sqsubseteq^*_{\mathcal{R}} p$, $Trans(q) = true$ and a path $s_0, \ldots, s_k$ s.t. $k \geq 1$, $s = s_0$, $o = s_k$ and $(s_i, s_{i+1}) \in \mathbf{E}(q)$ for $0 \leq i < k$*

Proof In [62] it is shown that the interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ defined from $\mathcal{T}$ as follows:

- $\Delta^{\mathcal{I}} = \mathbf{S}$
- $A^{\mathcal{I}} = \{s \mid A \in \mathcal{L}(s)$ for all atomic concepts A occurring in C or $\mathcal{O}\}$
- $p^{\mathcal{I}} = \begin{cases} \mathbf{E}(p)^{+} & \text{if Trans(p) = true} \\ \mathbf{E}(p) \cup \bigcup\limits_{\forall q, q \sqsubseteq^*_{\mathcal{R}} p} q^{\mathcal{I}} & \text{otherwise} \end{cases}$

is a model of $\mathcal{O}$. Moreover, it is shown that:

1. If $D \in \mathbf{L}(s)$ then $s \in D^{\mathcal{I}}$
2. $\langle s, t\rangle \in \mathbf{E}(p)$ iff $\langle s, t\rangle \in p^{\mathcal{I}}$ or there exists a role $q \sqsubseteq^*_{\mathcal{R}} p$ with $Trans(q) = true$ and a path $s_0, \ldots, s_k$ with $k \geq 1, s = s_0, t = s_k$ and $\langle s_i, s_{i+1}\rangle \in \mathbf{E}(q)$ for $0 \leq i < k$. Moreover, if p is simple, $p^{\mathcal{I}} = \mathbf{E}(p)$

Now, suppose that p is simple, $s \in \mathbf{S}, C \in \mathbf{L}(s)$ and $\langle s, o\rangle \notin \mathbf{E}(p)$ Using (1) and (2) above, we have that $s \in C^{\mathcal{I}}$ and $\langle s, o\rangle \notin p^{\mathcal{I}}$, which implies that $s \notin (\exists p.\{o\})^{\mathcal{I}}$.

Consequently, $\mathcal{I}$ is a model of $\mathcal{O}$ that does not satisfy the axiom $C \sqsubseteq \exists p.\{o\}$, and hence a contradiction.

Suppose that p is not simple and there is no path $s_0, \ldots, s_k$ with $k \geq 1, s = s_0, o = s_k$ and $\langle s_i, s_{i+1} \rangle \in \mathbf{E}(q)$ for $0 \leq i < k$ with $q \sqsubseteq^*_{\mathcal{R}} p$ and $Trans(q) = true$. If $C \in \mathbf{L}(s)$, then by (1) and (2), we have that $s \in C^{\mathcal{I}}$ and $\langle s, o \rangle \notin p^{\mathcal{I}}$, which again yields a contradiction. □

Lemma A.2 *Assume that there is a* simple *role p s.t. in every tableau $\mathcal{T} = (\mathbf{S}, \mathbf{L}, \mathbf{E})$ for C w.r.t. $\mathcal{O}$ if $C \in \mathbf{L}(s)$ for $s \in \mathbf{S}$ then $\langle s, o \rangle \in \mathbf{E}(p)$ where o is a nominal occurring in $\mathcal{O}$.*

Let $\mathbf{G} = (V, E, \mathcal{L}, \neq)$ be a clash-free and complete completion graph for C w.r.t. $\mathcal{O}$ and let the node $x \in V$ be s.t. $C \in \mathcal{L}(x)$.

Then, the nominal node $r_o \in V$ is a p-neighbor of x in $\mathbf{G}$.

Proof We will prove that from $\mathbf{G}$, which is clash free and complete, it is possible to construct a tableau $\mathcal{T}$ for C w.r.t. $\mathcal{O}$. The way this is done is identical to the soundness proof for $\mathcal{SHOIN}$ presented in [62].

More precisely, a *path* is a sequence of pairs of blockable nodes of $\mathbf{G}$ of the form $\tilde{p} = (\frac{x_0}{x'_0}, \ldots, \frac{x_n}{x'_n})$. For such a path we define $Tail(p) = x_n$ and $Tail'(\tilde{p}) = x'_n$. With $(\tilde{p}|\frac{x_{n+1}}{x'_{n+1}})$ we denote the path $\tilde{p} = (\frac{x_0}{x'_0}, ..., \frac{x_n}{x'_n}, \frac{x_{n+1}}{x'_{n+1}})$. The set $Paths(\mathbf{G})$ is inductively defined as follows:

- For each blockable node x of $\mathbf{G}$ that is a successor of a nominal node or a root node, $(\frac{x}{x}) \in Paths(\mathbf{G})$, and

- For a path $\tilde{p} \in Paths(\mathbf{G})$ and a blockable node y in $\mathbf{G}$:

  – If y is a successor of $Tail(\tilde{p})$ and y is not blocked, then $(p|\frac{y}{y}) \in Paths(\mathbf{G})$ and

  – If y is a successor of $Tail(\tilde{p})$ and y is blocked by y', then $(p|\frac{y'}{y}) \in Paths(\mathbf{G})$

Due to the construction of $Paths(\mathbf{G})$, all nodes occurring in a path are blockable and for $\tilde{p} \in Paths(\mathbf{G})$ with $\tilde{p} = (\tilde{p}'|\frac{x}{x'})$, x is not blocked, x' is blocked iff $x \neq x'$ and x' is never indirectly blocked. Furthermore the blocking condition implies $\mathcal{L}(x) = \mathcal{L}(x')$. We denote by $Nom(\mathbf{G})$ the set of nominal nodes in $\mathbf{G}$ and define a tableau $\mathcal{T} = (\mathbf{S}, \mathbf{L}, \mathbf{E})$ from $\mathbf{G}$ as follows:

- $\mathbf{S} = Nom(\mathbf{G}) \cup Paths(\mathbf{G})$
- $\mathbf{L}(\tilde{p}) = \begin{cases} \mathcal{L}(Tail(\tilde{p})) & \text{if } \tilde{p} \in Paths(\mathbf{G}) \\ \mathcal{L}(\tilde{p}) & \text{if } \tilde{p} \in Nom(\mathbf{G}) \end{cases}$
- $\mathbf{E}(R) = \{\langle \tilde{p}, \tilde{q} \rangle \in Paths(\mathbf{G} \times \mathbf{G}) \mid$

  $\tilde{q} = (p|\frac{x}{x'})$ and x' is an R-successor of $Tail(\tilde{p})$ or

  $\tilde{p} = (q|\frac{x}{x'})$ and x' is an inv(R)-successor of $Tail(\tilde{q})\} \cup$

  $\{\langle \tilde{p}, a \rangle \in Paths(\mathbf{G}) \times Nom(\mathbf{G}) |$ a is an R-neighbor of $Tail(\tilde{p})\} \cup$

  $\{\langle a, \tilde{p} \rangle \in Nom(\mathbf{G}) \times Paths(\mathbf{G}) |$ $\tilde{p}$ is an R-neighbor of $a\} \cup$

  $\{\langle a, b \rangle \in Nom(\mathbf{G}) \times Nom(\mathbf{G}) \mid$ b is an R-neighbor of $a\}$

In [62] it is proved that $\mathcal{T}$ constructed this way is a tableau for C w.r.t. $\mathcal{O}$.

Now, assume that the nominal node $r_o \in V$ is *not* a p-neighbor of x in $\mathbf{G}$ where $C \in \mathcal{L}(x)$. We show that we then encounter a contradiction.

There are three possibilities:

1. x is not blocked and is not a nominal node in $\mathbf{G}$
2. x is blocked and is not a nominal node in $\mathbf{G}$
3. x is a nominal node in $\mathbf{G}$

Suppose x is not a nominal node in $\mathbf{G}$, it is not blocked and $C \in \mathcal{L}(x)$. Since x is not a nominal node and it is not blocked then there is a path $\tilde{p}$ in $\mathbf{G}$ s.t. $Tail(\tilde{p}) = Tail'(\tilde{p}) = x$. By construction of $\mathcal{T}$, $\tilde{p} \in \mathbf{S}$ and $C \in \mathbf{L}(\tilde{p})$. By assumption of the Lemma, $\langle \tilde{p}, o \rangle \in \mathbf{E}(p)$. However, we also know that a is not a p-neighbor of $x = Tail'(\tilde{p})$ in $\mathbf{G}$ and by construction of $\mathcal{T}$, $\langle \tilde{p}, o \rangle \notin \mathbf{E}(p)$ and hence the contradiction.

Suppose x is not a nominal node in $\mathbf{G}$, it is blocked by y and $C \in \mathcal{L}(x)$. Since x is not a nominal node and it is blocked then there is a path $\tilde{p}$ in $\mathbf{G}$ s.t. $Tail(\tilde{p}) = y$ and $Tail'(\tilde{p}) = x$, with $\mathcal{L}(x) = \mathcal{L}(y)$. By construction of $\mathcal{T}$, $\tilde{p} \in \mathbf{S}$ and $C \in \mathbf{L}(\tilde{p})$. By assumption of the Lemma, $\langle \tilde{p}, o \rangle \in \mathbf{E}(p)$. However, we also know that $x = Tail'(\tilde{p})$ is not a p-neighbor of a in $\mathbf{G}$ and by construction of $\mathcal{T}$, $\langle \tilde{p}, o \rangle \notin \mathbf{E}(p)$ and hence the contradiction again.

Finally, suppose that x is a nominal node in $\mathbf{G}$ and $C \in \mathcal{L}(x)$. Since x is a nominal node then $x \in \mathbf{S}$ and $C \in \mathbf{L}(x)$, by construction of $\mathcal{T}$. By assumption of the Lemma, $\langle x, o \rangle \in \mathbf{E}(p)$. However, we also know that a is not a p-neighbor of x in $\mathbf{G}$ and by construction of $\mathcal{T}$, $\langle \tilde{p}, o \rangle \notin \mathbf{E}(p)$ and hence the contradiction again. □

Lemma A.3 *Assume that there is a* non-simple *role p s.t. in every tableau $\mathcal{T} = (\mathbf{S}, \mathbf{L}, \mathbf{E})$ for C w.r.t. $\mathcal{O}$ if $C \in \mathbf{L}(s)$, then there exists a role $q \sqsubseteq^*_{\mathcal{R}} p$ with $Trans(q) = true$ and a path $s_0, \ldots, s_k$ s.t. $k \geq 1$, $s = s_0$, $t = s_k$ and $(s_i, s_{i+1}) \in \mathbf{E}(q)$ for $0 \leq i < k$.*

Let $\mathbf{G} = (V, E, \mathcal{L}, \neq)$ be a clash-free and complete completion graph for C w.r.t. $\mathcal{O}$ and let the node $x \in V$ be a node with $C \in \mathcal{L}(x)$, then there exists a path $z_0, \ldots, z_k$ in $\mathbf{G}$ with $k \geq 1, x = z_0, o = z_k$ and z_i a q-neighbor of z_{i-1} for $0 \leq i < k$ and $q \sqsubseteq_{\mathcal{R}^*} p$.

Proof Let x be a node with $C \in \mathcal{L}(x)$, and assume that there is no path $z_0, \ldots, z_k$ in $\mathbf{G}$ with $k \geq 1, x = z_0, o = z_k$ and z_i a q-neighbor of z_{i-1} for $0 \leq i < k$ and $q \sqsubseteq_{\mathcal{R}^*} p$.

Identically to the proof of Lemma A.2, we can construct a tableau $\mathcal{T} = (\mathbf{S}, \mathbf{L}, \mathbf{E})$ from $\mathbf{G}$. By construction of $\mathcal{T}$, $C \in \mathcal{L}(\tilde{p})$, where $Tail(\tilde{p}) = x$. We have two possibilities:

- x is not an ancestor of o in $\mathbf{G}$.
- x is an ancestor of o, but there exists a pair of nodes y_1, y_2 s.t. x is an ancestor of y_1, y_2 is an ancestor of o and y_2 is a successor of y_1, but y_2 is not a q-neighbor of y_1.

In the first case, we obviously encounter a contradiction, because x and o are not even connected in $\mathbf{G}$. The second case reduces to the proof of Lemma 4. Let $\tilde{p}$, $\tilde{q}$ be paths in $\mathbf{G}$ (according to the definition of the set $Paths(\mathbf{G})$ in Lemma 4)

with $Tail'(\tilde{p}) = y_1$ and $Tail'(\tilde{p}) = y_2$ then $(\tilde{p}, \tilde{q}) \notin \mathbf{E}(q)$ (note that by construction $\tilde{p}, \tilde{q} \in \mathbf{S}$) and hence we find a contradiction. □

Theorem A.3 *Let $\mathcal{O} \models C \sqsubseteq \exists p.\{o\}$ with C satisfiable w.r.t. $\mathcal{O}$, then in every clash-free and complete graph $\mathbf{G}$ for C w.r.t. $\mathcal{O}$ there must exist a blockable node x with no predecessors (i.e. a root) that verifies the following:*

- *If p is simple then the nominal node o must be a p-neighbor of x in $\mathbf{G}$*
- *If p is not simple, then there must exist a path $z_0, \ldots, z_k$ in $\mathbf{G}$ with $k \geq 1, x = z_0, o = z_k$ and z_i a q-neighbor of z_{i-1} for $0 \leq i < k$ and $q \sqsubseteq^*_{\mathcal{R}} p$.*

Proof It is a straightforward consequence of the above lemmas. □

BIBLIOGRAPHY

[1] David W. Aha, Leonard A. Breslow, Héctor Muñoz-Avila, Dana S. Nau, and Rosina Weber. HICAP: Hierarchical interactive case-based architecture for planning, 1999.

[2] José Luis Ambite and Matthew Weathers. Automatic composition of aggregation workflows for transportation modeling. In *dg.o2005: Proceedings of the 2005 national conference on Digital government research*, pages 41–49. Digital Government Research Center, 2005.

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

[4] F. Baader, C. Lutz, M. Milicic, U. Sattler, and F. Wolter. Integrating description logics and action formalisms: First results. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, Pittsburgh, PA, USA, 2005.

[5] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[6] Franz Baader, Jan Hladik, Carsten Lutz, and Frank Wolter. From tableaux to automata for description logics. *Fundamenta Informaticae, 57:1-33*, 2003.

[7] Liviu Badea. Planning in description logics: Deduction versus satisfiability testing. In *Description Logics*, 1998.

[8] B. Benatallah and F. Casati. Special issue on web services. *Distributed and Parallel Databases*, 12(2–3), September 2002.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[10] Jim Blythe, Ewa Deelman, and Yolanda Gil. Automatically composed workflows for grid environments. *IEEE Intelligent Systems*, 19(4):16–23, 2004.

[11] Anthony J. Bonner and Michael Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[12] David Booth, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. W3C Working Group Note 11 February 2004 http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[13] Alex Borgida. On the relationship between description logic and predicate logic queries. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 219–225, New York, NY, USA, 1994. ACM Press.

[14] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. Classic: a structural data model for objects. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 58–67, New York, NY, USA, 1989. ACM Press.

[15] D. Brickley and R.V. Guha. RDF vocabulary description language: RDF schema. W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[16] D. Brickley and R.V. Guha. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation submitted 22 February 1999 http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[17] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. In *Proc. of the 13th Int. World Wide Web Conference (WWW 2004)*, May 2004.

[18] F. Casati, D. Georgakopoulos, and M. Shan. Special issue on e-services. *VLDB Journal*, 24(1), January 2001.

[19] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[20] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Classic planning for mobile robots. In *Proceedings of the FAPR-96 Workshop on Planning in Complex Environments*, 1996.

[21] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Description logic-based framework for planning with sensing actions. In *Proceedings of the 1997 Description Logic Workshop (DL'97)*, pages 39–43, 1997.

[22] Giuseppe De Giacomo and Maurizio Lenzerini. Boosting the correspondence between description logics and propositional dynamic logics. In *Proceedings of AAAI-94, 12th Conference of the American Association for Artificial Intelligence*, pages 205–212, Seattle, US, 1994.

[23] Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 http://www.w3.org/TR/2002/WD-owl-ref-20021112/.

[24] Premkumar T. Devanbu and Diane J. Litman. Taxonomic plan reasoning. *Artif. Intell.*, 84(1-2):1–35, 1996.

[25] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Al-log: integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10:227–252, 1998.

[26] Kutluhan Erol, James Hendler, and Dana S. Nau. Htn planning: complexity and expressivity. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, pages 1123–1128. American Association for Artificial Intelligence, 1994.

[27] Kutluhan Erol, James Hendler, and Dana S. Nau. Semantics for HTN planning. Technical Report CS-TR-3239, University of Maryland at College Park, 1994.

[28] Oren Etzioni, Keith Golden, and Daniel Weld. Tractable closed world reasoning with updates. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *KR'94: Principles of Knowledge Representation and Reasoning*, pages 178–189. Morgan Kaufmann, San Francisco, California, 1994.

[29] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 88–97. Kaufmann, San Mateo, CA, 1990.

[30] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open world planning in the situation calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 754–760, Menlo Park, CA, July 2000. AAAI Press.

[31] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[32] Maria Fox and Derek Long. International planning competition, 2002. http://www.dur.ac.uk/d.p.long/competition.html.

[33] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms.* PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[34] Andrew Garland, Kathy Ryall, and Charles Rich. Learning hierarchical task models by defining and refining examples. In *K-CAP '01: Proceedings of the*

1st international conference on Knowledge capture, pages 44–51, New York, NY, USA, 2001. ACM Press.

[35] Michael R. Geneserth and Richard E. Fikes. Knowledge Interchange Format Version 3.0 Reference Manual. Technical Report Logic Group Report Logic-92-1, Computer Science Department, Stanford University, June 1992.

[36] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*, chapter Hierarchical Task Network Planning. Morgan Kaufmann, 2004.

[37] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[38] M I Ginsberg and M L Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–80, 1986.

[39] M. L. Ginsberg and D. E. Smith. Reasoning about action i: A possible worlds approach. *Artificial Intelligence*, 35:165–195, 1988.

[40] K. Golden. *Planning and Knowledge Representation for Softbots*. PhD thesis, University of Washington, 1997.

[41] K. Golden, O. Etzioni, and D. Weld. Planning with execution and incomplete information. Technical Report TR96-01-09, Department of Computer Science, University of Washington, February 1996.

[42] Robert Gombotz and Schahram Dustdar. On web services workflow mining. In *Business Process Management Workshops*, pages 216–228, 2005.

[43] Javier Gonzalez-Castillo, David Trastour, and Claudio Bartolini. Description Logics for Matchmaking of Services. In *Workshop on Applications of Description Logics ADL 2001*, Vienna, 2002.

[44] B. Cuenca Grau, B. Parsia, and E.Sirin. Combining OWL ontologies using $\mathcal{E}$-connections. *Journal of Web Semantics*, 4(1), January 2005.

[45] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework - W3C Recommendation 24 June 2003, 2001. http://www.w3.org/TR/soap12-part1/.

[46] V. Haarslev and R. Möller. An empirical evaluation of optimization strategies for abox reasoning in expressive description logics. In *Proc. of the Int. Description Logics Workshop (DL'99)*, pages 115–119, 1999.

[47] V. Haarslev, R. Möller, and A.Y. Turhan. Exploiting pseudo models for TBox and ABox reasoning in expressive description logics. In *IJCAR 2001, Italy*, 2001.

[48] Volker Haarslev and Ralf Mller. Optimization techniques for retrieving resources described in owl/rdf documents: First results. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *Proceedings of the*

Ninth International Conference Principles of Knowledge Representation and Reasoning (KR2004), pages 163–174, Whistler, Canada, 2004. AAAI Press.

[49] J. Heflin and H. Munoz-Avila. Lcw-based agent planning for the semantic web, 2002.

[50] J. Heinsohn, D. Kudenko, B. Nebel, and H.-J. Profitlich. Rat - representation of actions using terminological logics. In J. Heinsohn and B. Hollunder, editors, *DFKI Workshop on Taxonomic Reasoning*, 1992. number D-92-08 in DFKI Document.

[51] J. Hendler. Agents and the semantic web. *IEEE Intelligent Systems*, 16(2), 2001.

[52] Andreas Herzig and Omar Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, 115(1):107–138, 1999.

[53] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[54] I. Horrocks. Implementation and optimisation techniques. In *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003.

[55] I. Horrocks and P. F. Patel-Schneider. DL systems comparison. In *International Description Logics Workshop (DL'98)*, pages 55–57, 1998.

[56] I. Horrocks and U. Sattler. Ontology reasoning in the $\mathcal{SHOQ}(\mathcal{D})$ description logic. In *Proc. of the 17th Int. Joint Conf. on AI (IJCAI)*, pages 199–204, 2001.

[57] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[58] I. Horrocks and S. Tessaris. A conjunctive query language for description logic aboxes. In *Proc. of the 17th Nat. Conf. on Artificial Intelligence (AAAI 2000)*, pages 399–404, 2000.

[59] Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM, 2004.

[60] Ian Horrocks and Ulrike Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999.

[61] Ian Horrocks and Ulrike Sattler. Optimised reasoning for $\mathcal{SHIQ}$. In *Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002)*, pages 277–281, July 2002.

[62] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for SHOIQ. In *Proc. of the 19th Int. Joint Conf. on AI (IJCAI 2005)*. Morgan Kaufman, 2005.

[63] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for very expressive description logics. *J. of the Interest Group in Pure and Applied Logic*, 8(3):239–264, 2000.

[64] Ian Horrocks and Sergio Tessaris. Querying the semantic web: a formal approach. In Ian Horrocks and James Hendler, editors, *Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science, pages 177–191. Springer-Verlag, 2002.

[65] Ian Horrocks and Sergio Tessaris. Querying the semantic web: a formal approach. In *Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002)*, 2002.

[66] Okhtay Ilghami, Dana S. Nau, and Hector Muñoz-Avila. Learning to do HTN planning. In *Proceedings of the Sixteenth International Conference on AI Planning and Scheduling*, Cumbria, UK, June 2006. AAAI Press.

[67] Okhtay Ilghami, Dana S. Nau, Hector Muñoz-Avila, and David W. Aha. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence*, 21(4):388–413, november 2005.

[68] Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Planning with sensing, concurrency, and exogenous events: Logical framework and implementation. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 678–689, San Francisco, 2000. Morgan Kaufmann.

[69] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[70] Aditya Kalyanpur. *Debugging and Repair of OWL Ontologies*. PhD thesis, University of Maryland, 2006.

[71] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In P. Gardenfors, editor, *Belief Revision*, volume 29 of Cambridge Tracts in Theoretical Computer Science, pages 183–203. Cambridge University Press, 1992.

[72] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[73] Christel Kemke. A formal approach to describing action concepts in taxonomical knowledge bases. In *Foundations of Intelligent Systems, 14th International Symposium (ISMIS)*, pages 657–662, 2003.

[74] Christel Kemke. A formal theory for describing action concepts in terminological knowledge bases. In *Canadian Conference on AI*, pages 458–465, 2003.

[75] Ugur Kuter, Evren Sirin, Dana Nau, Bijan Parsia, and James Hendler. Information gathering during planning for web service composition. In *Proceedings of 3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, November 2003.

[76] O. Kutz, C. Lutz, F. Wolter, and M. Zakharyaschev. $\mathcal{E}$-Connections of Abstract Description Systems. *Artificial Intelligence 156(1):1-73*, 2004.

[77] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[78] Lei Li and Ian Horrocks. A Software Framework For Matchmaking Based on Semantic Web Technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*, Budapest, Hungary, May 2003.

[79] Thorsten Liebig and Dietmar Rösner. Action hierarchies in description logics. In *Description Logics Workshop*, 1997.

[80] Fangzhen Lin and Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[81] H. Liu, C. Lutz, M. Milicic, and F. Wolter. Updating description logic aboxes. In *Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, Lake District, UK, 2006.

[82] Erick Martinez and Yves Lespérance. Web service composition as a planning task: Experiments using knowledge-based planning. In *Proceedings of the ICAPS-2004 Workshop on Planning and Scheduling for Web and Grid Services*, pages 62–69, June 2004.

[83] Matthew Mason. Kicking the sensing habit. *AI Magazine*, 14(1):58–59, 1993.

[84] Ryusuke Masuoka, Bijan Parsia, and Yannis Labrou. Task computing - the semantic web meets pervasive computing -. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, Florida, October 2003.

[85] D. McDermott. Drs: A set of conventions for representing logical languages in rdf. http://www.daml.org/services/owl-s/1.0/DRSguide.pdf, 2004.

[86] D. McDermott and D. Dou. Representing disjunction and quantifiers in RDF. In I. Horrocks and J. Hendler, editors, *ISWC 2002*, volume 2342, pages 250–263, 2002.

[87] Drew McDermott. Estimated-regression planning for interactions with web services. In *AIPS*, pages 204–211, 2002.

[88] Sheila McIlraith and T. Son. Adapting Golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, Toulouse, France, apr 2002.

[89] B. Motik and U. Sattler. Practical DL reasoning over large ABoxes with KAON2. In *Proc. KR-2006*, 2006.

[90] Boris Motik. *Reasoning in Description Logics using Resolution and Deductive Databases*. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany, January 2006.

[91] Boris Motik, Ulrike Sattler, and Rudi Studer. Query answering for owl-dl with rules. *Journal of Web Semantics*, 3(1):41–60, JUL 2005.

[92] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, may 2002.

[93] D. Nau, T.C. Au, O. Ilghami, U. Kuter, J.W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[94] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, Seattle, aug 2001.

[95] Dana S. Nau, Yue Cao, Amnon Lotem, and Hector Muñoz-Avila. Shop: Simple hierarchical ordered planner. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 968–975, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[96] OWL Services Coalition. OWL-S: Semantic markup for web services, 2003. OWL-S White Paper http://www.daml.org/services/owl-s/0.9/owl-s.pdf.

[97] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *The First International Semantic Web Conference*, 2002.

[98] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Importing the semantic web in uddi. In *CAiSE '02/ WES '02: Revised Pa-*

pers from the International Workshop on Web Services, E-Business, and the Semantic Web, pages 225–236, London, UK, 2002. Springer-Verlag.

[99] Michael P. Papazoglou. Web services and business transactions. *World Wide Web*, 6(1):49–91, 2003.

[100] Edwin P. D. Pednault. ADL: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332. Morgan Kaufmann Publishers Inc., 1989.

[101] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, pages 103–114. Kaufmann, San Mateo, CA, 1992.

[102] J. S. Penberthy and D. S. Weld. Ucpop: A sound, complete, partial order planner for adl. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, pages 103–114. Kaufmann, San Mateo, CA, 1992.

[103] R. Petrick and F. Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *In Proceedings of the Fourth International Conference on AI Planning and Scheduling (AIPS'98)*, pages 86–93, 1998.

[104] Marco Pistore, Fabio Barbon, Piergiorgio Bertoli, Dmitry Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *The 11th International Conference on Artificial Intelligence, Methodologies, Systems, and Applications (AIMSA)*, pages 106–115, 2004.

[105] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, 2001.

[106] R. Reiter. *Knowledge In Action: Logical Foundations for Specifying and Implementing Dynamical Systems.* The MIT Press, 2001.

[107] Raymond Reiter. On specifying database updates. *Journal of Logic Programming*, 25(1):53–91, 1995.

[108] U. Sattler. A concept language extended with different kinds of transitive roles. In B. Nebel, editor, *Proc. of the 20th German Annual Conf. on Artificial Intelligence (KI 2001), Vol. 1137 of Lecture Notes In Artificial Intelligence*, pages 199–204. Springer Verlag, 2001.

[109] A. Schaerf. Reasoning with individuals in concept languages. *Data and Knowledge Engineering*, 13(2):141–176, 1994.

[110] Klaus Schild. A correspondence theory for terminological logics: preliminary report. In *Proceedings of IJCAI-91, 12th International Joint Conference on Artificial Intelligence*, pages 466–471, Sidney, AU, 1991.

[111] Kaarthik Sivashanmugam, John A. Miller, Amit P. Sheth, and Kunal Verma. Framework for semantic web process composition. *International Journal of Electronic Commerce*, 9(2):71, Winter 2004-05.

[112] M.K. Smith, C. Welty, and D.L. McGuiness. OWL Web Ontology Language Guide. W3C Recommendation http://www.w3.org/TR/owl-guide/, 2004.

[113] Paolo Traverso and Marco Pistore. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*, pages 380–394, 2004.

[114] UDDI. The UDDI technical white paper, 2000. http://www.uddi.org/.

[115] Jeffrey D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[116] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[117] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of Lecture Notes in Computer Science, pages 18–29, 2000.

[118] T. Weijters and W.M.P. van der Aalst. Process mining: Discovering workflow models from event-based data. In B. Krse, M. de Rijke, G. Schreiber,

and M. van Someren, editors, *Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC 2001)*, pages 283–290, 2001.

[119] Marianne Winslett. Sometimes updates are circumscription. In *IJCAI*, pages 859–863, 1989.

[120] Marianne Winslett. *Updating logical databases*. Cambridge University Press, New York, NY, USA, 1990.

[121] B. Kiepuszewski W.M.P. van der Aalst, A.H.M. ter Hofstede and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[122] WS-Policy. Web services policy framework (ws-policy). http://www-106.ibm.com/ developerworks/library/specification/ws-polfram/.

[123] WS-Trust. Web services trust language (ws-trust). http://www-128.ibm.com/ developerworks/library/specification/ws-trust/.

[124] Qiang Yang. Formalizing planning knowledge for hierarchical planning. *Comput. Intell.*, 6(1):12–24, 1990.

[125] Zhengxiang Pan Yuanbo Guo and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2):158–182, 2005.