Knowledge Representation and Concretization of Underdetermined Data

Vegar Skaret

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Knowledge Representation and Concretization of Underdetermined Data

Vegar Skaret

Knowledge Representation and Concretization of Underdetermined Data

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Data is underdetermined when there are variables whose values are unknown and the legal instantiations of these variables depend on given factors and the corresponding domain knowledge. The problem of instantiating the variables is a combinatorial problem. In this thesis we develop an approach to find all of the legal instantiations based on a specific knowledge formalization. We make use of a popular knowledge representation tool to make the formalization of knowledge for use cases with underdetermined data more accessible to users without programming experience. We argue that the approach is easier to use and to maintain than an alternative approach formalizing knowledge with imperative programming.

We formalize a way of representing underdetermined data with an OWL ontology, and develop an algorithm to generate all of the legal instantiations based on the ontology. We also show a way of translating worlds represented in OWL to a representation of worlds in the Maude system. Furthermore, we argue for correctness of the algorithm by proving that it generates sound results and give the intuition for, and partly prove, how it generates complete results. The approach is successfully applied on parts of a use case of petroleum systems. However, the execution time of the implementation is long, and this is likely due to poor computational complexity. We discuss possible approaches to mitigate this issue.

Contents

List of Figures

Listings

Preface

This project has been a little journey. It started as an attempt at having Maude and Prolog interact during a Maude computation to have one reasoner for dynamic processes and another for static knowledge.[1] This got shelved, but in the meantime I hacked together a Python script as a temporary solution to a problem in the Geological Assistant.[2] Then, one of my supervisors, Leif Harald, suggested that we try to solve the same problem using the Web Ontology Language (OWL). From that suggestion, the topic for the thesis was settled.

Acknowledgements

I have been extremely fortunate to write my thesis in collaboration with the Geological Assistant project at the SIRIUS research center.[3] Not only did this result in a project I found interesting, but the people in the Geological Assistant team have been very competent, helpful, inclusive and a joy to work with. From my first day in the project I felt like an active participant of it, and this has made the time working on the thesis so much more exciting.

There are several people I would like to thank for their contributions to the thesis project and to the thesis itself. First and foremost, I am deeply grateful to my main supervisor Crystal Din and co-supervisors Leif Harald Karlsen and Ingrid Yu. They have all been dedicated to the project, happy to discuss any problems and given invaluable feedbacks and suggestions along the way. In particular, Crystal has been of tremendous help with the writing of the thesis, Leif Harald has been a great source of knowledge on OWL and Description Logic, and he has had a big impact on the thesis by discussing and giving suggestions for the algorithm proposed, and Ingrid Yu has given important guidance to the thesis work. Without their help, I cannot imagine how I would be able to produce this thesis.

[1] `https://github.com/vskaret/mapl-com`

[2] `https://github.com/vskaret/PythonScenarioExpansion`

[3] `https://sirius-labs.no/geological-assistant/`

Geology is the application domain of the thesis, and as a computer scientist with no experience in geology, it has been difficult to get a grasp of the terminology and concepts. I thank Irina Pene for being of great help for understanding the geology, and for always being happy to explain a concept even when it is the tenth time doing so. Also thanks to Fabrício Rodrigues who had a great suggestion for the algorithm proposed, which is that it might be usable for OWL ontology debugging. All of the members of the Geological Assistant team must be thanked for including me in the project, working alongside these researchers has been a great learning experience.

I thank my friends Michael and Kai for a great many tips, all sorts of discussions, and some epic sessions of table tennis during the time of the thesis work. I thank as well my good friend Vis for encouraging me to pursue a master's degree. I am grateful to my roommates, my brother, Lars, and Anne Catherin, for letting me use the dining room table as an office for a great part of the writing of the thesis, and for being kind landlords. Lastly, a special thanks go to my parents: my father, Bjørn, for encouraging me to get an education to be able to work with something I enjoy, and my mother, Anita, for always being supportive.

Part I

Preliminaries

Chapter 1

Introduction

Knowledge representation [12] is a part of computer science focusing on formalizing knowledge. It has its roots in mathematical logic, and has evolved into modeling of many different parts of the world. This thesis is done as a part of a project in the SIRIUS center for research-driven innovation.[1] The center's research focus is on how to make use of data generated in the oil and gas industry. In the thesis we look at formalizing knowledge about a type of geological data which is underdetermined, and enabling the use of the data by generating all of its legal instantiations based on the knowledge formalized. Our goal is to do this with the help of the Web Ontology Language.

1.1 Underdetermination of Data

Underdetermination is a form of uncertainty, and we use the term specifically for situations where the values of one or more variables are unknown, but it is known what the possible values for those variables are. For example, a human's blood type can be either A, B, AB or O. If one does not know the blood type of a person, it is a case of underdetermination as the variable blood type's value is unknown and its possible values are among these four cases. Due to the nature of underdetermination, the increase of underdetermined variables causes a combinatorial expansion to the possible situations. For example, if there is one person for whom the blood type is unknown, there are four possible situations. However, if there are two people, X_1 and X_2, the number is already increased to 16 as shown in the following sets.

$$\{A_1, A_2\}, \{A_1, B_2\}, \{A_1, AB_2\}, \{A_1, O_2\},$$
$$\{B_1, A_2\}, \{B_1, B_2\}, \{B_1, AB_2\}, \{B_1, O_2\},$$

[1] `https://sirius-labs.no`

$$\{AB_1, A_2\}, \{AB_1, B_2\}, \{AB_1, AB_2\}, \{AB_1, O_2\},$$
$$\{O_1, A_2\}, \{O_1, B_2\}, \{O_1, AB_2\}, \{O_1, O_2\}$$

where A_1 means X_1 has blood type A and so on.

A medical doctor gathering information about a patient is an example of concretization of underdetermined data. If a person feels sick and goes to the doctor, the doctor usually asks questions and takes tests in the task of finding a diagnosis. For example the patient's lifestyle habits such as exercise, diet, consumption of narcotics, alcohol, smoking, etc. might affect how the doctor evaluates the situation. Then there are values that are found through tests like blood tests and x-rays. Through questions and tests, the doctor gathers information to make values of variables that was unknown, known. We use the term to *concretize* for the act of making the value of an unknown variable concrete, either through gathering information or making an assumption. All of the concrete variables in combination helps the doctor in narrowing down the range of possible diseases when assigning a diagnosis.

The probabilities of the different values a variable can have is not always quantifiable. In the case of blood types it probably is. However, it is difficult to quantify the probability of the existence of another intelligent species in the universe, but we do know that either it exists or it does not exist. In this thesis, we do not quantify probabilities of the different values, but view them as different possibilities. As a consequence, our work is more qualitative than quantitative, and the range of possibilities are made to help and inspire ideas for experts in their analyses.

1.2 Geological Assistant

The Geological Assistant is an interdisciplinary research project at the SIRIUS center for research and innovation.[2] The main goal of the project is to develop a system to support explorationists in their analyses [6]. To achieve this, the system models static geological data and dynamic geological processes to generate plausible scenarios of how an area of interest ended up like it is today. The scenarios are possible histories of events. Due to underdetermination of data, instead of capturing uncertainties with probability, we represent discrete scenarios as branches of potential alternatives.

Currently the modeling is done within the domain of petroleum geology. An issue in this domain is the lack of information about what the world looks like today due to hydrocarbons being located below the surface. Further, it is not always certain what processes led to the current state of affairs. What is even more uncertain is what

[2] `https://sirius-labs.no/geological-assistant`

the situation was at a point of interest in the past. One might know some things, but it is very rare to have all facts about a case. Furthermore, gathering information can be expensive, requiring drilling deep below the surface to fetch physical samples. Thus, when reasoning about the potential for accumulation somewhere, there are many variables with unknown values, where the possible values are limited.

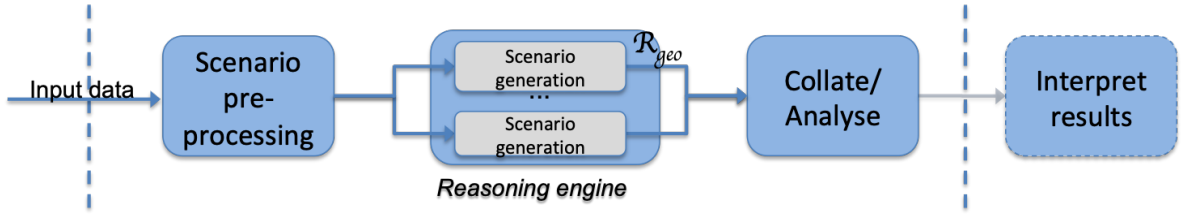


Figure 1.1: Geological Assistant pipeline [6]

The workflow of the Geological Assistant system is visualized in Figure 1.1. There are three main parts of the Geological Assistant Pipeline. **The first part is scenario pre-processing, which, based on the given domain knowledge, generates all of the possible proto-scenarios from the underdetermined input data. This part of the pipeline is where this thesis work contributes to.** A proto-scenario is one concretization of the underdetermined input data, and it serves as an initial state for the further scenario generation located in the second stage of the pipeline. A scenario is produced by simulation and consists of a sequence of execution states. The last part of the pipeline is to collate, analyse, and interpret the generated scenarios.

1.3 Contributions

The scope of the thesis is to develop a methodology for knowledge representation of underdetermined data to be used in further reasoning of dynamic processes. The methodology is applied on a specific use case of petroleum geology, which is exploration of new petroleum deposits in a depositional environment known as ancient submarine fan. A model is made to represent the static data of the use case, such as geological entities and their spatial relations. We call such a model a world, and a world is underdetermined if the value of any attribute of its entities is not known. If everything about the world is known, we say it is concrete. These concepts are defined more precisely in Chapter 4.

The research is divided into the following questions.

1. Research Question 1 (RQ1) How can underdetermined worlds be represented within the Web Ontology Language (OWL)?

2. Research Question 2 (RQ2) How can all of the legal concretizations of a world be generated using OWL?

3. Research Question 3 (RQ3) How can a representation of a world be translated from one language to another, specifically from OWL to Maude?

To help answering RQ1 we have the following goals.

1.1. Specify what underdetermined worlds are and formalize an abstract way of representing them.

1.2. Develop a methodology using OWL axioms and assertions to represent worlds.

1.3. Formalize the knowledge of a petroleum exploration use case in an OWL ontology.

To answer RQ2 we aim to do the following.

2.1. Develop an algorithm which based on an OWL representation of an underdetermined world generates all of the world's concretizations.

2.2. Show that the algorithm generates sound and complete results.

2.3. Implement the algorithm and test it on the ontology of a petroleum exploration use case.

For RQ3 our goals are as follows.

3.1. Make use of the abstract representation from RQ1 to guide the translation of OWL entities to Maude entities.

3.2. Implement a converter using the Maude system.

In the conclusion of the thesis in Section 8.2 on page 92 we detail how the questions posed here are answered in the thesis text.

1.4 Research Method

As this thesis work is part of an innovation research project exploring an application area of knowledge representation and building a software system, our approach to the research is a combination of the formal methodology and "build" methodology as detailed in [7].

We use these two methodologies because we want to inspect whether using a logical framework to solve a problem within the Geological Assistant software is

possible. We formalize the problem we investigate, and we formalize an algorithm that is capable of solving the problem. Then we build a software module as a proof of concept to confirm that the approach is in fact applicable to solve the problem. Furthermore, it allows us to evaluate the practical utility of the approach developed.

1.5 Thesis Overview

Chapter 1 Introduction to the thesis, the context it is set in and its contributions.

Chapter 2 The technical background of the thesis: Description Logic, the Web Ontology Language and the Maude programming language.

Chapter 3 Presentation of the petroleum exploration use case domain.

Chapter 4 Abstract problem formalization. The chapter serves as an introduction to the problem of the thesis work.

Chapter 5 Methodology for solving the problem. An approach for generating concrete worlds by using OWL is presented. First, how to model the worlds defined in chapter 4 with Description Logic is shown. Then, an algorithm capable of generating the concrete worlds of an OWL ontology is presented. Lastly, the correctness of the results of the algorithm is argued for and its complexity is briefly analyzed.

Chapter 6 Application of the methodology. The world representation and concretization approach from Chapter 5 is applied to the use case presented in Chapter 3 to build a module that can be used as a part of the Geological Assistant.

Chapter 7 Discussion and evaluation of the approach developed and the use case results.

Chapter 8 The final chapter of the thesis. It contains a review of related work, a summary of how the research questions were answered and a brief discussion of future work.

Chapter 2

Technical Background

A knowledge representation tool known as the Web Ontology Language[1] and its theoretical foundation Description Logic forms the basis for this thesis work of scenario pre-processing, i.e. proto-scenario generation. In this chapter we give an introduction to these concepts. As this thesis work is part of the Geological Assistant project where the Maude system is used for further scenario generation, a short introduction to Maude syntax is also included.

2.1 Description Logic and the Web Ontology Language

Description Logic (DL) [3, 20, 34] is a method of representing knowledge through knowledge bases. A knowledge base is a formalization of knowledge about a domain through axioms, often separated into three boxes of axiom types: terminologic (TBox), relational (RBox) and assertional (ABox). In the TBox, concepts (unary relations) of the universe are named and described. How the roles (binary relations) of the universe work and relate to each other is described in the RBox. In the ABox, a specific world is instantiated by assigning concepts and roles to individuals (elements of the universe).

DL is a family of languages that are fragments of first-order logic [41]. An important characteristic of these languages is that most of them are decidable for important reasoning problems such as satisfiability. Decidable means that the process will always halt, meaning it will never loop (run infinitely). Their expressiveness and computational complexities are important research topics within the DL research community.

DL is a widely applied form of knowledge representation by being used as the logical foundations for a Semantic Web language known as the Web Ontology

[1] `https://www.w3.org/OWL/`

Language (OWL) [26, 11]. OWL is an active research topic, and it has a decent amount of tool support. The tools supported include an open-source GUI called Protégé[2] [29], the OWL API[3] [14] for working with OWL 2 ontologies and popular reasoners such as HermiT [10], Pellet [40] and FaCT++ [44]. OWL has been successfully applied in areas such as biological and biomedical research [13] and in improving the accuracy of searching for Quranic verses [46].

We start off this section by presenting a DL which is often extended upon, called $\mathcal{ALC}$. Then, we show how one can increase the expressivity of $\mathcal{ALC}$ to get to the most expressive DL of OWL 2: $\mathcal{SROIQ}$ [16]. We then present the open-world semantics used in DL and OWL, and show how the DL $\mathcal{ALC}$ can be expressed in OWL terms. Lastly, we briefly present the different languages available in OWL 2.

2.1.1 A Description Logic

Syntax

The language $\mathcal{ALC}$, Attributive Language with Complements, is the core of some important DLs and its syntax is as follows.

$$
\begin{array}{rcll}
\text{C, D} & \longrightarrow & \text{A} \mid & \text{(atomic concept)} \\
 & & \top \mid & \text{(universal concept)} \\
 & & \bot \mid & \text{(bottom concept)} \\
 & & C \sqcap D \mid & \text{(intersection)} \\
 & & C \sqcup D \mid & \text{(union)} \\
 & & \neg C \mid & \text{(negation)} \\
 & & \forall R.C \mid & \text{(value restriction)} \\
 & & \exists R.C & \text{(existential quantification)}
\end{array}
$$

where C and D are complex concepts and R is an atomic role. The syntactical elements making it possible to construct larger entities such as intersection are known as constructs.

Semantics

The semantics of a DL language is formalized through all possible interpretations $\mathcal{I}$. The interpretations have two elements: a non-empty set Δ and an assignment function $\cdot^{\mathcal{I}}$. Δ is the universe of elements, and the elements are known as individuals. The assignment function assigns all individual names $a^{\mathcal{I}}$ to an element in $\Delta^{\mathcal{I}}$, all atomic concepts $A^{\mathcal{I}}$ to a subset of $\Delta^{\mathcal{I}}$ and all roles $R^{\mathcal{I}}$ to a binary relation subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The semantics of $\mathcal{ALC}$ is as follows.

[2] `https://protege.stanford.edu/`

[3] Source code and wiki at `https://github.com/owlcs/owlapi`

$$
\begin{aligned}
\top &= \Delta^{\mathcal{I}} \\
\bot &= \emptyset \\
(C \sqcap D)^{\mathcal{I}} &= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} &= \left\{ a \in \Delta^{\mathcal{I}} \mid \forall b.(a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}} \right\} \\
(\exists R.C)^{\mathcal{I}} &= \left\{ a \in \Delta^{\mathcal{I}} \mid \exists b.(a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \right\}
\end{aligned}
$$

where a and b are individuals of the universe Δ.

Most of the semantics are self-explanatory, i.e. $\top$ is always interpreted to the set that contains all of the individuals, and $\bot$ is always interpreted to the empty set. The most advanced aspects are the value restriction $\forall R.C$ and existential quantification $\exists R.C$. The semantics of the value restriction means that an individual satisfies the restriction if all elements it is R-related to are also of the concept C. In other words, an individual is member of the set if it is only R-related to individuals of type C, or if it is not R-related to any individuals at all. The existential quantification simply states that the individual is R-related to at least one individual of type C.

Axioms

The syntax defines what can be expressed syntactically with the language, while axioms make use of the syntax to express statements. In $\mathcal{ALC}$ there are TBox and ABox axioms, meaning one can express statements about the concepts and individuals of the knowledge base. Even though RBox axioms are missing, roles can still be used as a part of TBox axioms such as the value restriction.

The TBox axioms are concept inclusion and concept equivalence. Their syntax and semantics are as follows.

TBox Axioms	Syntax	Semantics
concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
concept equivalence	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$

Inclusion is also known as subsumption. With concept inclusion one can state that the members of one concept also are members of another concept, for example that a defensive midfielder in football is a type of midfielder.

$$DefensiveMidfielder \sqsubseteq Midfielder$$

With concept equivalence, one can go even further and state that the members of two concepts are exactly the same, for example that a football player is either a goalkeeper, defender, midfielder or attacker.

$$Footballer \equiv Goalkeeper \sqcup Defender \sqcup Midfielder \sqcup Attacker$$

The ABox axioms are concept assertion, role assertion, individual equality and individual inequality, and they are formalized as follows.

ABox Axioms	Syntax	Semantics
concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$R(a,b)$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
individual equality	$a \approx b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$
individual inequality	$a \not\approx b$	$a^{\mathcal{I}} \neq b^{\mathcal{I}}$

The ABox axioms are used to assert that something is true, they are similar to the tuples of databases. An ABox is formally a finite set of ABox axioms. With concept assertions one can make statements about the concepts of individuals. For example, "Haaland is an attacker" can be asserted with

$$Attacker(haaland),$$

where *Attacker* is a concept and *haaland* is an individual.

With role assertions one can make statements about how an individual relates to other individuals. For example, the statement "Haaland plays for Dortmund" can be asserted as follows.

$$playsFor(haaland, dortmund),$$

where *playsFor* is a role representing that a football player plays for a football club.

With individual equality and inequality axioms, one can state that two individuals are equal or not equal. This means that the unique name assumption, where it would be assumed that each individual name represents its own individual, is not a part of the language. For example, one can state that "Haaland and Oedegaard are not the same individual":

$$haaland \not\approx oedegaard$$

Now we have presented the syntax and axioms available in the specific DL $\mathcal{ALC}$, in the next section we look at expansions to $\mathcal{ALC}$. These add more constructors and axioms resulting in the ability to express more statements.

2.1.2 Increasing Expressivity

The language $\mathcal{ALC}$ is the core of a family of languages which expands upon it, and each expansion has its own letter to identify it. There has been much research involved with developing languages with different levels of expressivity while maintaining the lowest level of reasoning complexity possible. In general, the more expressive a language becomes, the more complex the reasoning becomes. Now we

describe the constructors and axioms necessary to add for the language $\mathcal{SROIQ}^{(\mathcal{D})}$, which roughly corresponds to the direct semantics of OWL 2.

To go from $\mathcal{ALC}$ to $\mathcal{SROIQ}$, we start with going from $\mathcal{ALC}$ to $\mathcal{SR}$ by adding complex role inclusions. After which we add the three expansions $\mathcal{O}$ (nominal concepts), $\mathcal{I}$ (inverse roles), $\mathcal{Q}$ (arbitrary qualified number restrictions) and $(\mathcal{D})$ (datatypes). The order of which these are added does not matter, and adding one of them to $\mathcal{ALC}$ by itself represents a legitimate DL, for example adding $\mathcal{Q}$ results in the DL $\mathcal{ALCQ}$. We now present these extensions and their formalities.

Complex role inclusions and more: $\mathcal{SR}$

Complex role inclusions adds the support for stating the RBox axiom with the following syntax and semantics to $\mathcal{ALC}$.

$$(R_1 \circ R_2 \sqsubseteq S)^{\mathcal{I}} = R_1^{\mathcal{I}} \circ R_2^{\mathcal{I}} \sqsubseteq S^{\mathcal{I}}$$

where R and S are roles and $\circ$ is role composition. An important restriction on role composition is that it can only be on the left-hand side of complex role inclusions.

With this axiom, one can express transitivity such as the boss of my boss is also my boss

$$bossOf \circ bossOf \sqsubseteq bossOf$$

In other words, if one has the statements $bossOf(a, b)$ and $bossOf(b, c)$, then one can infer that $bossOf(a, c)$.

Further, $\mathcal{SR}$ adds the self concept which can be used to relate an individual to itself, for example to express reflexivity of a role such as everyone thinks of themselves

$$\top \sqsubseteq \exists thinksOf.Self,$$

and irreflexivity such as no one attends their own funeral

$$\top \sqsubseteq \neg\exists attendsFuneralOf.Self.$$

Lastly, $\mathcal{SR}$ adds six role assertions, also called role characteristics, where one of them adds expressivity:

$$Disjoint(R, S) = R^{\mathcal{I}} \cap S^{\mathcal{I}} = \emptyset$$

With this assertion, one can express that two roles are disjoint. For example that elements of a role expressing the *older than* relation is disjoint from the elements of the role expressing the *younger than* relation. With the addition of all of this expressivity to $\mathcal{ALC}$, the name of the DL is changed to $\mathcal{SR}$.

Nominals: $\mathcal{O}$

With nominals the capability of declaring closed concepts is added. A closed concept is a concept where all individuals that are part of the concept is specified. It has the following syntax and semantics.

$$(\{a_1, ..., a_n\})^{\mathcal{I}} = \{a_1^{\mathcal{I}}, ..., a_n^{\mathcal{I}}\}$$

For example the days of the weekend can be expressed as the concept

$$WeekendDay \equiv \{saturday, sunday\}$$

Note that nominals blurs the line between the TBox and ABox somewhat, as named individuals which are instantiated in the ABox are used to describe concepts in the TBox. With nominals added to $\mathcal{SR}$, the name of the DL is changed to $\mathcal{SRO}$.

Inverse roles: $\mathcal{I}$

With inverse roles, the capability of turning the order of the elements in a role around is added. This means that

$$(R^-)^{\mathcal{I}} = \{(a, b) \mid (b, a) \in R^{\mathcal{I}}\}$$

With this axiom, one can express symmetry such that if a is colleague of b, then b is also colleague of a

$$colleagueOf \equiv colleagueOf^-,$$

and asymmetry such as one cannot be the boss of one's boss

$$Disjoint(bossOf, bossOf^-),$$

With inverse roles added to $\mathcal{SRO}$, the name of the DL is changed to $\mathcal{SROI}$.

Qualified number restrictions: $\mathcal{Q}$

$\mathcal{Q}$ adds two constructors $\geqslant n\, R.C$ (at-least restriction) and $\leqslant n\, R.C$ (at-most-restriction), where n is a range of natural numbers. For an individual to satisfy the at-least restriction, it must be R-related to at least n individuals of type C. While to satisfy the at-most restriction, it must be R-related to at most n individuals of type C. The at-least restriction is formally interpreted as follows.

$$(\geqslant n\, R.C)^{\mathcal{I}} = \left\{ a \in \Delta^{\mathcal{I}} \;\middle|\; \left| \left\{ b \mid (a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \right\} \right| \geqslant n \right\}$$

where "$|\cdot|$" denotes set cardinality.

With the at-least restriction, one can express that a person must at least have one birthplace

$$Person \sqsubseteq \geqslant 1\ bornIn.Place,$$

while the at-most restriction is interpreted as:

$$(\leqslant n\ R.C)^{\mathcal{I}} = \left\{ a \in \Delta^{\mathcal{I}} \;\middle|\; \left| \left\{ b \mid (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \right\} \right| \leqslant n \right\}$$

With the at-most restriction, one can express that a person can at most have one birthplace

$$Person \sqsubseteq \leqslant 1\ bornIn.Place,$$

and this specific axiom makes the *bornIn* role functional for the *Person* concept. With qualified number restrictions added to $\mathcal{SROI}$, the name of the DL is changed to $\mathcal{SROIQ}$.

Datatypes: $(\mathcal{D})$

With $(\mathcal{D})$, the support for having concrete domains as a part of the knowledge base is added. A concrete domain is a set of pre-defined individuals such as integers. It specifically makes it possible to have individuals from concrete domains as the second argument of roles known as concrete roles.

With a concrete role, one can for example state the age of an individual

$$age(Alfred, 51),$$

where 51 is in the concrete domain of natural numbers.

With datatypes added to $\mathcal{SROIQ}$, the name of the DL is changed to $\mathcal{SROIQ}^{(\mathcal{D})}$. This is an important language as it is the basis of the direct semantics of OWL 2.

Role Axioms Summary

With the increased expressivity, axioms for roles are added. They are the following.

RBox Axioms	Syntax	Semantics
role inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
role equivalence	$R \equiv S$	$R^{\mathcal{I}} = S^{\mathcal{I}}$
complex role inclusion	$R_1 \circ R_2 \sqsubseteq S$	$R_1^{\mathcal{I}} \circ R_2^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
role disjointness	$Disjoint(R,S)$	$R^{\mathcal{I}} \cap S^{\mathcal{I}} = \emptyset$

Role inclusion and equivalence works the same as concept inclusion and equivalence, only that they are used for roles instead of concepts. Examples of complex role inclusion and role disjointness were given above.

2.1.3 Open-world Semantics

An important property of DL knowledge bases is that they use open-world semantics. This is the opposite of closed-world semantics usually used in relational databases. With closed-world semantics, it is not possible to infer new knowledge in the knowledge base.

To explain the open-world semantics, we first look at what closed-world semantics is by reviewing the semantics of regular databases. A database is similar to a knowledge base. It has a schema with terminological knowledge and a set of tuples populating the schema with assertional knowledge. However, a big difference is that a database only has one satisfying interpretation due to the closed-world semantics. This means that what you see is what you get, there is for example no implicit knowledge a reasoner can infer. Anything not asserted to be true is by definition false. There is a negation by absence.

With an open-world assumption, however, missing information is not interpreted to being false. It is just interpreted to being missing, not known whether it is true or not. Thus an ABox has several models. The benefit of using the open-world assumption is that it is then possible to use reasoners to infer new knowledge. If there is some fact which is true in all of the models in the ABox, but it is not stated explicitly, the reasoner can make this knowledge explicit.

2.1.4 Reasoning Tasks

Thus far we have mostly showed the formalities of DL and what a knowledge base consists of. Now we get to the exciting part of DL, which is to utilize its formalities to automatically perform tasks such as inferring new knowledge with a reasoner. Reasoners are devices that connect to the knowledge base and use search algorithms to syntactically prove various things about the axioms, for example that the knowledge base is consistent (satisfiable). Now we briefly present some of the most important reasoning tasks for DLs.

Knowledge Base Consistency

The knowledge base KB, which is a set of TBox, RBox and ABox axioms, is consistent if is satisfiable. More specifically, if there is an interpretation $\mathcal{I}$ such that $\mathcal{I} \models KB$, where $\models$ denotes "makes satisfiable" or "models". In other words, there must be at least one interpretation that satisfies all of the axioms of the knowledge base. If there is no such interpretation, the knowledge base is inconsistent.

Assume a knowledge base KB_F has the following TBox T_F, stating that a football game consists of football players and referees, and that a football player cannot be a referee.

$$FootballGame \equiv Footballer \sqcup Referee$$
$$Footballer \sqsubseteq \neg Referee$$

Having the following ABox A_F axioms would make a consistent knowledge base, as all of the concepts and concept assertions are satisfiable.

$$Footballer(carew)$$
$$Referee(mohn)$$

However, adding the concept assertion $Footballer(mohn)$ to A_F would make the knowledge base KB_F inconsistent, as any interpretation $\mathcal{I}$ such that $\mathcal{I} \models Footballer(mohn)$ and $\mathcal{I} \models Referee(mohn)$ will not $\mathcal{I} \models Footballer \sqsubseteq \neg Referee$.

Concept Consistency

The reasoning of a concept C's consistency is constrained to the TBox of an knowledge base KB. For C to be consistent in KB, there must be an interpretation $\mathcal{I}$ such that $\mathcal{I} \models KB$ and $C \neq \emptyset$.

Axiom Entailment

A knowledge base KB entails a DL axiom α if every model of KB is also a model of α. In other words, if there is an interpretation which satisfies KB without satisfying α, KB does not entail α.

Instance Retrieval

Instance retrieval is the task of retrieving all individuals of a concept C from a knowledge base KB. It has two restrictions, (i) only individuals named in KB are retrieved and (ii) the individuals retrieved must be an instance of C in all of KB's models.

Classification

The concepts in the TBox of a knowledge base KB can be ordered in a hierarchy according to the orders of the subsumption relationship. This hierarchy of concepts will have $\top$ as the root and $\bot$ as all of the leaves. For example, the $DefensiveMidfielder$ concept would be a child of the $Midfielder$ concept in such a hierarchy. In OWL, this hierarchy is known as the class hierarchy.

2.1.5 From Description Logic to OWL

Everything which is expressable in DL is also expressable in OWL. However, OWL uses different terminology and syntax than DL. The following tables shows the corresponding DL and OWL terminologies.

DL Term		OWL Term
concept	$\rightarrow$	class
role	$\rightarrow$	property
knowledge base	$\rightarrow$	ontology
axiom	$\rightarrow$	axiom
individual	$\rightarrow$	individual

OWL supports several syntaxes, for example the Manchester Syntax[4] [15]. What follows is the $\mathcal{ALC}$ DL syntax translated to Manchester Syntax. $\top \rightarrow$ owl:Thing is read as "top translates to owl:Thing".

DL		OWL
$\top$	$\rightarrow$	owl:Thing
$\bot$	$\rightarrow$	owl:Nothing
$C \sqcap D$	$\rightarrow$	C and D
$C \sqcup D$	$\rightarrow$	C or D
$\neg C$	$\rightarrow$	not C
$\forall R.C$	$\rightarrow$	R only C
$\exists R.C$	$\rightarrow$	R some C

Because the OWL API is used for the implementation in this thesis, from now on we use OWL terminology. However, for the sake of easy representation of ontologies and DL semantics we will use the DL syntax for axioms and use the concepts of TBox, RBox and ABox axioms.

2.1.6 OWL Profiles

In this section we briefly present the motivations and complexities of three OWL 2 profiles [27] which are fragments of the Direct Semantics. The Direct Semantics of OWL 2 corresponds with the presented DL $\mathcal{SROIQ}^{(D)}$, and it is often called OWL 2 DL. There are three profiles available in OWL 2, and these are OWL 2 EL, OWL 2 QL and OWL 2 RL. Common with all of them is that they trade expressiveness in return for better computational complexity. An important thing for this thesis is that only OWL 2 DL is capable of expressing disjoint union of concepts. Why this is important will become more clear in Chapter 5.

[4] `https://www.w3.org/TR/owl2-manchester-syntax/`

OWL 2 EL

OWL 2 EL corresponds to the DL $\mathcal{EL}^{++}$ [2]. It is most useful for knowledge bases with many concepts and/or roles that do not require more constructors than intersection, existential quantification and datatypes. The complexity of reasoning tasks except conjunctive query answering are all in polynomial time.

OWL 2 QL

OWL 2 QL corresponds to DL-Lite$_{\mathcal{R}}$ [4]. This profile is developed for use cases where one is interested in querying knowledge bases with large ABoxes. It was chosen as a profile because sound and complete query answering is in LOGSPACE when measured with the size of the ABox used as input size.

OWL 2 RL

OWL 2 RL corresponds to the language of Description Logic Prover (DLP) [33] which is a superset of propositional dynamic logic [8]. It is the most expressive profile while remaining polynomial time complexity for most reasoning tasks, except when reasoning with the whole knowledge box as an input. Then it is either in co-NP or NP depending on the task.

2.2 Maude

The Maude system [5] is a rewrite engine capable of executing rewrite theories, and it is commonly used for verification of systems and processes. It has its own declarative programming language which is designed for modeling systems and processes. We do not elaborate on the theoretical parts of rewriting logic, but we do show most of the Maude syntax. We first show how the natural numbers can be defined in Maude and how the + function can be added. Then we briefly present the Maude built-ins configurations, messages and objects in combination with rewrite rules.

2.2.1 Functional Modules

A module in Maude is a program, and as the name suggests it can be a part of larger programs. There are two types of modules, functional modules `fmod` and system modules `mod`. A functional module works much like most functional programming languages, except that the functions are not of higher order. It consists mainly of the elements sorts and subsorts (types and subtypes), constants, relations, equations and variables.

A functional module describes something static. It manipulates a state by using equations to reduce the state until no more equations can be applied. The resulting state is called a ground term. An equation is a function where the input is equivalent to the output. Thus, a functional module can not perform an operation where a state transitions into state which is not equivalent to the state at the beginning. For this, rewrite rules are needed which are defined in system modules. A functional module will always terminate if specified correctly.

The code for a functional module is located between `fmod` and `endfm` which declares the start and end of a functional module. The functional module `MY-FUNCTIONAL-MODULE` is declared as follows.

```
fmod MY-FUNCTIONAL-MODULE is
endfm
```

2.2.2 Sorts

Sorts are declared with the `sort` keyword, and they are names of datatypes. In the snippet below we declare the sort `Nat`, which we use for natural numbers.

```
sort Nat .
```

Subtyping is declared with `subsort Nat < Int`, where `Nat` is a subsort of `Int`.

```
sort Int .
subsort Nat < Int .
```

2.2.3 Constants and Functions

Constants are declared with the following syntax.

```
op constant : -> s [props] .
```

where `constant` is the name of the constant, `s` is the sort of the constant and `props` are properties of the operator. The `[]` signifies that giving `props` is optional in the syntax. However, when `props` are specified, the operator must satisfy the properties assigned. An example constant is the natural number 0 which can be declared as follows.

```
op 0 : -> Nat [ctor] .
```

As 0 is a constant of the natural numbers, it is declared as a `Nat`. Functions are declared with the following syntax.

```
op function : s_1, ..., s_n -> s [props] .
```

where `function` is the function name, s_1 to s_n are the function arguments and their order, and `s` is the sort of the function. The rest of the natural numbers can be defined with the successor function. For example with the operator `s` which takes one natural number as input and returns a natural number:

```
op s : Nat -> Nat [ctor] .
```

With `0` and `s` defined this way, `0` is the first natural number, `s(0)` is the second natural number, and so on. The [ctor] means that the operator is a constructor, which is an element of the data type of its sort. For example `0` and `s(s(0))` are elements of `Nat`.

A non-constructor function is a function that operates on constants and other functions. Their semantics are formalized with equations, and equations are declared with the `eq` keyword.

```
op _+_ : Nat Nat -> Nat .
eq 0 + M = M .
eq s(M) + N = s(M + N) .
```

In the snippet above the + function is formalized. The declaration states that there are two parameters. Note the underlines in `_+_` lets one use infix notation (`0 + 0`) as well as prefix notation (`+(0, 0)`) when calling the function. There are also conditional equations, which will be executed only if its conditions are satisfied:

```
ceq term = term' if c1 /\ ... /\ cn .
```

where c_i are booleans and `/\` is conjunction.

The equations use variables `M` and `N`. These work similarly to variables in mathematical functions, in the sense that they cannot destructively change their value as in imperative programming. Variables are declared using the `var` and `vars` keywords, where the only difference is that with `var` only one variable can be declared while with `vars` one or more can be declared. `M` and `N` can be declared as variables of sort `Nat` as follows:

```
vars M N : Nat .
```

2.2.4 An Example of a Functional Module

All of the previous snippets can be collected in the functional module `NAT-ADD` as shown in Figure 2.1. This makes it possible to run + calculations in the Maude console with the `red` (reduce) command. The `red` keyword will apply equations to the term until no more are applicable and it terminates. For example "`red 0 + s( s(0)).`" returns "`result Nat: s(s(0))`". The Maude system finds which equations are applicable with pattern matching.

```
1  fmod NAT-ADD is
2    sort Nat .
3    op 0 : -> Nat [ctor] .
4    op s : Nat -> Nat [ctor] .
5    op _+_ : Nat Nat -> Nat .
6
7    vars M N : Nat .
8
9    *** Define the addition function recursively:
10   eq 0 + M = M .
11   eq s(M) + N = s(M + N) .
12 endfm
```

Figure 2.1: Functional module implementing + in Maude [31]

Modules can be used within other modules by importing with keywords `protecting` or `including`. For example, `NAT-ADD` can be imported to `NAT-LIST` as follows.

```
fmod NAT-LIST is
  protecting NAT-ADD .
endfm
```

2.2.5 System Modules

A system module is a specification of a rewrite theory. The main difference between system modules and functional modules is the additions of rewrite rules which lets one model dynamic systems. The rewrite rules works with pattern matching as equations. A system module is declared with `mod` and ended with `endm`:

```
mod MY-SYSTEM-MODULE is
endm
```

2.2.6 Configurations, Messages, Objects and Rewrite Rules

`Configuration`, `Msg` and `Object` are built-in sorts imported from the module `CONFIGURATION`. A configuration represents a specific state of objects and messages. An often used analogy is to view the configuration as a soup. Within the soup are messages and objects floating around, and the messages are concurrently read, removed and created by the objects. Further, the objects can change their state depending on the content of a message.

The sorts `Msg` and `Object` are subsorts of `Configuration`. While `Msg` and `Configuration` are only sorts, `Object` is a built-in function with a special syntax which is as follows.

`< o : C | a1: v1, ..., an: vn >`

where `o` is the identifier of the object, `C` is the class name, a_i are attribute names and v_i are attribute values. This syntax makes it easier to read an object as an object is of a specific class, and all of its attributes are named. An object is declared as follows.

`op <_ : C | a1:_, ..., an:_ > : Oid, s1, ..., sn -> Object [ctor] .`

where `C` and a_i are as above, `Oid` is the built-in sort for object identifier and s_i are the sorts of a_i, respectively.

We now show an example module `FIRESTATION` with the example configuration `fireconfig` of a firestation with a fireman on call responding to a fire.

```
1  mod FIRESTATION is
2    protecting STRING .
3    protecting NAT .
4    protecting CONFIGURATION .
5
6    sorts Firestation Fireman Status Alert .
7    subsorts Firestation Fireman < Object .
8    subsorts Fire Alert < Msg .
9    subsorts String < Oid.
10
11   op <_: Firestation | workers:_ > : Oid Nat -> Object .
12   op <_: Fireman | workplace:_, status:_ > : Oid Oid Status -> Object .
13   op fire : Nat -> Fire .
14   op fireman-alert : Oid Nat -> Alert .
15   op on-call : -> Status .
16   op busy : Nat -> Status .
17
18   op fireconfig: -> Configuration .
19   eq fireconfig = < "Houston" : Firestation | workers: 1 >
20   < "Frank" : Fireman | workplace: "Houston", status: on-call >
21   fire(0) .
22 endm
```

The `Firestation` object has an attribute for the amount of workers. The `Fireman` object has one attribute for his workplace and another attribute for his status. Then there is a `fire` function with a `Nat` parameter representing its identifier. A fireman is either `on-call` or `busy` with a fire. The `busy` function has a `Nat` argument representing the fire the fireman is busy extinguishing.

The `fireconfig` is a `Configuration` consisting of one `Firestation Object`, one `Fireman Object` and one `Fire Msg`. A `Fire` can be part of a `Configuration` because it is set to be a subsort of `Msg`. The `Firestation` has `Oid` "Houston". A string can be used as `Oid` due to `String` being set as a subsort of `Oid`. The fire station has the attribute `workers` which states how many firemen is on call. Then there is a `Fireman` named `"Frank"` who works at the `"Houston" Firestation` and is ready to work.

The fire station needs a way to respond to the fire and issue the fire extinguishing mission to its worker. To achieve this, we can use rewrite rules. Rewrite rules are different from equations in that the result is not necessarily equivalent to the input and new states are generated.

The syntax of rewrite rules `rl`, and conditional rewrite rules `crl`, looks as follows.

```
rl [label] : t => t' .

crl [label] : t => t' if condition .
```

The `t` is the pattern of some of the terms in the configuration before executing the rule, and `t'` the pattern after the rule is executed. The label is the name of the rule and is optional, and the condition works exactly the same as for conditional equations. The following listing shows 3 rewrite rules made to react to a `fire`.

```
1  vars O O' O'' : Oid .
2  vars N N' N'' : Nat .
3
4  crl [firestation-respond] :
5  < O : Firestation | workers: N > fire(N')
6    =>
7      < O : Firestation | workers: N-1 > fire(N') fireman-alert(O, N')
8      if N > 0 .
9
10 rl [fireman-respond] :
11 < O : Fireman | workplace: O', status: on-call > fireman-alert(O', N)
12   =>
13     < O : Fireman | workplace: O', status: busy(N) > .
14
15 rl [fire-extinguish] :
16 < O : Fireman | workplace: O', status: busy(N) > fire(N)
17 < O' : Firestation | workers: N' >
18   =>
19     < O : Fireman | workplace: O', status: on-call >
20     < O' : Firestation | workers: N'+1 > .
```

First, in the `firestation-respond` rule a firestation with available firemen responds to a fire by issuing an alert to its firemen on call at the station. Second, a fireman takes on the extinguishing mission with the `fireman-respond` rule. Third, a fireman extinguishes a fire and goes back to the fire station in `fire-extinguished`.

In the `fireconfig` example, `firestation-respond` is executed first because it is the only rewrite rule which matches the pattern of `fireconfig`. It will make the configuration change from

```
< "Houston" : Firestation | workers: 1 >
< "Frank" : Fireman | workplace: "Houston", status: on-call >
fire(0) .
```

to

```
< "Houston" : Firestation | workers: 0 >
< "Frank" : Fireman | workplace: "Houston", status: on-call >
fireman-alert("Houston", 0) fire(0) .
```

Then the fireman takes on the mission through `fireman-respond`, setting his status to `busy(0)` representing that he is busy extinguishing `fire(0)`.

```
< "Houston" : Firestation | workers: 0 >
< "Frank" : Fireman | workplace: "Houston", status: busy(0) >
fire(0) .
```

Finally, `fire-extinguished` is executed, with the resulting configuration.

```
< "Houston" : Firestation | workers: 1 >
< "Frank" : Fireman | workplace: "Houston", status: on-call > .
```

In this state, nothing more can happen because none of the rewrite rules matches with the state of the configuration. However, rewritings can be infinite. For example, if we add an `Arsonist` object which starts fires and never stops doing it, the computation will go on infinitely.

This ends the presentation of the technical background, in the next chapter we present the application domain of the thesis. The most important part of the Maude chapter is how configurations are built.

Chapter 3

Application Domain

In the previous chapter we presented the technical background for the thesis. In this chapter we present the domain we apply our tool on, which is petroleum systems from geology. We give a brief high-level introduction to the topic. As the thesis is mostly focused on a methodology for generating results with a model, we keep the model simple.

The first part of this section introduces the geology of petroleum as a whole, then we go into some specifics of petroleum systems, and end the chapter with presenting a petroleum system model based on synthetic data. We use many terms that are specific to geology and petroleum geology in particular, and those that are important to our work are explained as we go along.

3.1 Petroleum Geology

Petroleum [21, 17] is a term commonly used for all forms of hydrocarbons, which is a composition of hydrogen and carbons. Some of its forms are crude oil, natural gas and asphalt. Hydrocarbons have been an important energy resource since the nineteenth century [43]. As hydrocarbons are generated far below the surface by organic material which was once at the surface of the earth, the full process of generating hydrocarbons is very slow. Further, hydrocarbons do not regenerate when they are burned for fuel. Thus, hydrocarbons are a limited resource, and until energy sources which can replace hydrocarbons are found, continual discovery of new sources of hydrocarbons are needed for our energy-demanding society. Naturally, it becomes more and more difficult to find petroleum as more of it has been extracted, so it is important to explore and test new ideas and methods in the search for more of it.

A petroleum explorationist works with finding new deposits of petroleum. Hydrocarbons accumulate below the surface, thus searching for potential accumula-

tions often requires working with unknown factors. “Each pool is unique - we may think of a pool as the end result of twenty or twenty-five variables of which only a few can be ascertained in advance” [21]. A test drilling is done to secure evidence of the physical properties of a subsurface area, which might include the existence of hydrocarbons. However, test drilling is expensive. Some factors that can be ascertained before a test drilling are the properties interpretable in seismic images, which are generated through an echo location technique similar to sonar, radar and ultrasound [22].

3.2 Petroleum Systems

Petroleum system [24, 23] is an essential geological concept for the oil and gas industry. It is what describes the entities and processes that must be in place for hydrocarbons to exist in a specific location. No petroleum system, no hydrocarbons, and vice versa. In this section we present a simplified version of the concept.

A petroleum system requires two sets of factors: (i) essential elements and (ii) processes. The study of the first considers examining the current geological state of the world, while the study of the second is more about hypothesizing on historical events and processes that led to the current state of the world.

The essential elements are:

- Source rock
- Reservoir rock
- Seal rock

The source rock is the system’s source of hydrocarbons. Reservoir rocks are capable of storing hydrocarbons partly because they are porous and permeable. Porosity states the amount of void space there is in a rock, and permeability states the ability of a porous material to have fluids move through it. Seal rocks are rocks which hydrocarbons are unable to flow through.

The processes of the petroleum systems are:

- Trap formation
- Hydrocarbon generation, migration and accumulation

A trap is an area sealed off from migration where hydrocarbons can accumulate. Figure 3.1 on the facing page shows a drawing of oil and gas accumulated in an anticlinal trap. Since the hydrocarbons move towards lower pressure, they will usually go as far up as possible. A concave reservoir rock with a top seal as in the drawing is thus a location where there potentially is an accumulation.

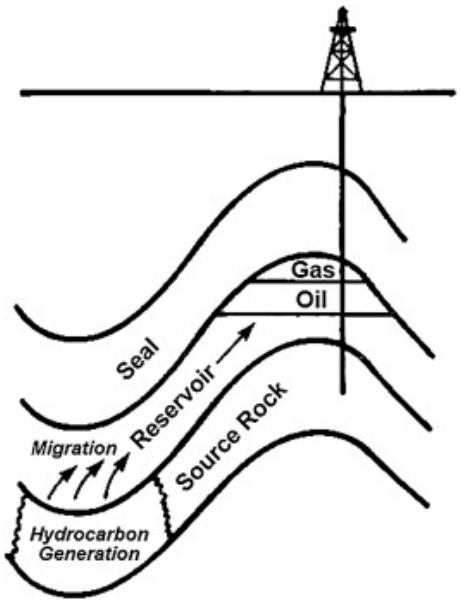


Figure 3.1: A petroleum system [36]

There are several types of traps. A fault, which is a discontinuity in a body of rock, can form a lateral seal as shown in Figure 3.2. However, a fault does not necessarily always function as a seal as they can also leak hydrocarbons. When they do leak, the hydrocarbons are not trapped if the leak leads to another reservoir or seep.

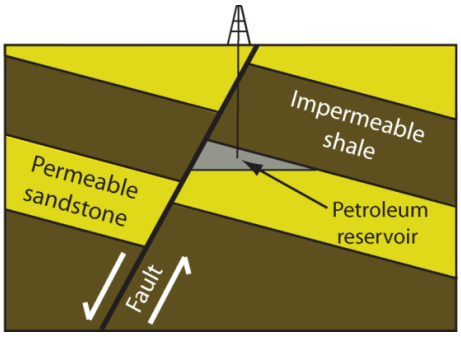


Figure 3.2: A fault trap [1]

Hydrocarbon generation, migration and accumulation are three distinct processes: Firstly, generation is the process of hydrocarbons coming to existence. This happens in source rocks, where organic material (carbon) is "cooked" at high temperatures due to high pressure, fusing carbons with hydrogen.

Secondly, migration is the process of hydrocarbons moving. At first, they move out of the source rock. Later, they migrate towards a trap or a seep. Migrations from traps can happen due to a trap being filled to its spill point. Imagine more and more hydrocarbons accumulating in the trap in Figure 3.1, at one point there will be no more room which means some hydrocarbons will be pushed out of the trap and migrate further. Another reason for migrations from traps is a seal rock losing its sealing capacity.

Thirdly, accumulation is the process of hydrocarbons massing in a location due to being unable to migrate any further. An accumulation always happens inside of a trap. The alternative to accumulation is a seep, where hydrocarbons have migrated all the way to the surface.

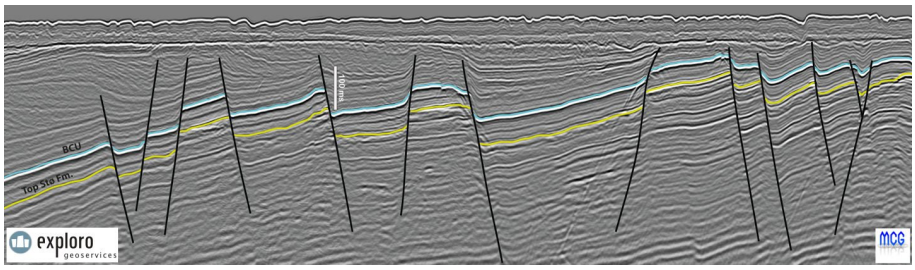


Figure 3.3: An interpreted seismic image [9]

An important part of an explorationist's analysis of discovering petroleum systems is to interpret seismic images. Figure 3.3 shows an example of a seismic image, where one can see structural information about geological layers below the surface. Each layer was on the earth's surface at different geological eras. This image has been interpreted by an explorationist as can be seen by the colored horizontal and black vertical lines.

In summary, a petroleum system needs a source rock, a trap and a pathway from the source rock to the trap the hydrocarbons can migrate through. Thus if a test drilling is done, and hydrocarbons are extracted, it is proof of the existence of a petroleum system. A petroleum system can have several locations of accumulation, so even if the accumulation at the test location was not large enough for commercialization, it might be a clue that leads to the discovery of larger accumulations in the surrounding area. All subsurface data gathered are clues that helps in solving the mystery of potentially locating an accumulation which is profitable to extract.

3.3 Exploration Use Case

Now that we have introduced the concept of petroleum system, we will look at an application of the concept for a potential petroleum system located in an ancient submarine fan environment. Our use case entails modeling of a petroleum system where we assume some knowledge of the entities in the system. However, not everything is known, and these unknown factors are relevant in determination of whether or not there is an accumulation of hydrocarbons.

The goal of the complete use case exercise as detailed in [6] is to find all possible current states of the system given a geological model, and give geological reasons for a state being as it is. For example, if there is not an accumulation in a geological unit (abbreviated GeoUnit or GU) that has the potential to be a trap, it should be able to explain why it is not the case. One explanation could be that hydrocarbons never reached the trap because of another trap. Another could be that due to missing a lateral seal it was not a trap after all, and the migration continued further. These explanations of how the results are generated can be important to explorationists to see the effects of different assumptions.

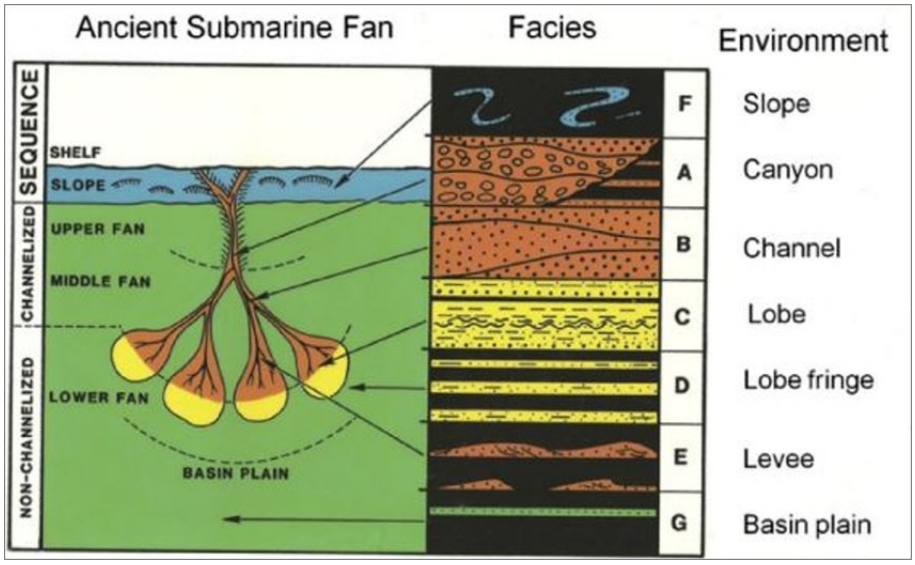


Figure 3.4: Ancient submarine fan depositional environment. Figure originally in [38], and is based on concepts introduced in [30]

We use the following definition of submarine fan: "Channel and lobe (or sheet sand) complexes formed from sediment-gravity flows in the deep-sea environment, commonly beyond the continental shelf" [37]. As can be seen in Figure 3.4, there are 7 facies A-G in a submarine fan, and a facies is a body of rock with specific

properties. For example, some facies can be seal rock, while others can be reservoir rock. Furthermore, the different facies are located at different areas in the submarine fan.

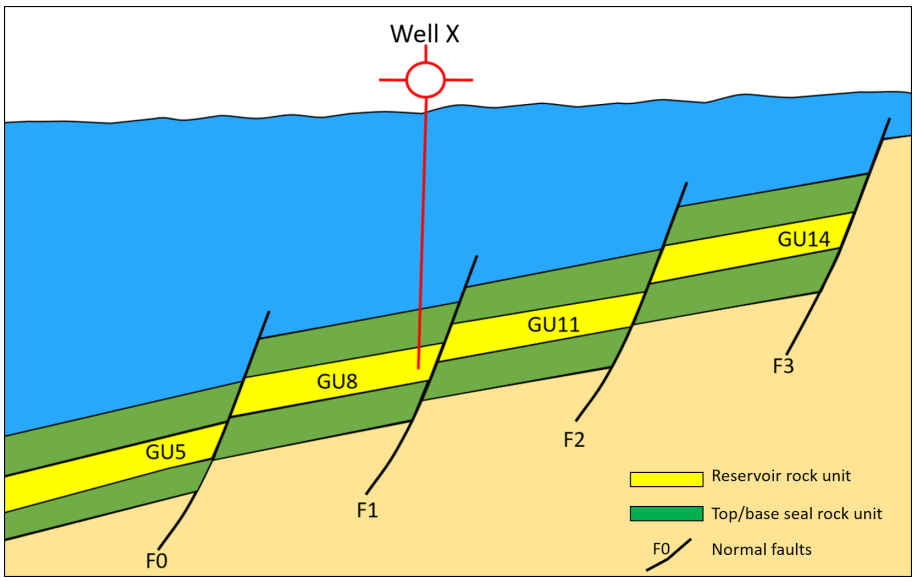


Figure 3.5: The use case [6]

Figure 3.5 shows a 2D graphic model of the use case. It details the possible petroleum system comprising 4 fault blocks representing the possible traps. Each block being made up of 3 geological units representing source, seal and reservoir rocks. Each fault block is bounded by a normal fault. The rocks in the fault blocks are geological units named GU_i, where i is the unique identifier of the unit. A consequence of the layer discontinuity is that for the hydrocarbons to be able to migrate from one reservoir rock to another, the normal fault to its right (point of lowest pressure) cannot be sealing.

In the use case for this thesis work we assume there are two types of important attributes of the geological units that are unknown. First, the sealing capacities of all of the faults F_0, F_1, F_2 and F_3 are unknown. Second, we assume whether or not the reservoir rocks GU_5, GU_8, GU_{11} and GU_{14} are in fact reservoir rocks is unknown, and it depends on their location in the submarine fan. In some locations, like in a feeder channel, the facies are porous and permeable and thus allows hydrocarbon migration. In others, like in a basin plain, they are neither porous nor permeable, so no hydrocarbons can migrate through them.

3.4 Use Case Model

The thesis work revolves around finding the different instantiations of the use case described above, given a simplified domain model we formalize in this section. In particular, we are interested in the faults F_0, F_1, F_2 and F_3, and the rock units GU_5, GU_8, GU_{11} and GU_{14}. We do not have complete knowledge of these entities' important attributes. We assume we know everything we need about the other entities of the use case.

First, it is not known what the sealing capacities of the faults are. We assume they can either be sealing or non-sealing. This gives the following set of values for fault sealing capacity

$$SealingCapacity = \{Sealing, NonSealing\}.$$

Second, it is unknown whether GU_5, GU_8, GU_{11} and GU_{14} are reservoir rocks. To determine that a rock unit is a reservoir rock, we must first find what depositional environment it is located in as portrayed in Figure 3.4. To model the depositional environments, we use the locations *FeederChannel* (*FC*), *InterChannel* (IC_1 and IC_2), *DistributaryChannel* (*DC*), *Lobe* (*L*), *LobeFringe* (*LF*) and *BasinPlain* (*BP*). The reason for having IC_1 and IC_2 is that an *InterChannel* can come both before and after a *DistributaryChannel*. IC_1 represents an inter channel located before a distributary channel, and IC_2 represents an inter channel located after a distributary channel. We assume all geological units located in the following areas are porous and permeable, and thus capable of being reservoir rocks: *FeederChannel*, *DistributaryChannel* and *Lobe*. The others are seal rocks.

The depositional environments of a submarine fan are in an order which depends on which side of the submarine fan one starts. We assume we start at the slope and end in the basin plain. Thus the *FeederChannel* located in the slope is the first entity in the model. A unit located in a *FeederChannel* can only be in front of a unit also located in a *Feederchannel*, or located in an $InterChannel_1$ or in a *DistributaryChannel*. We denote this as

$$FeederChannel \rightarrow \{FeederChannel, InterChannel_1, DistributaryChannel\}.$$

This is read as *FeederChannel* can be in front of *FeederChannel*, $InterChannel_1$ or *DistributaryChannel*. All of the orderings are shown in Figure 3.6.

With the sealing capacity and depositional environment ordering formalizations, one interpretation of the use case is that all of the faults F_0, F_1, F_2 and F_3 are non-sealing, and all of the rock units GU_5, GU_8, GU_{11} and GU_{14} are located in a feeder channel. Because they are located in a feeder channel, we know that they are all porous and permeable. In this thesis, we want to find all possible instantiations of the use case based on the knowledge presented here.

$$
\begin{aligned}
FeederChannel &\rightarrow \{FeederChannel, InterChannel_1, DistributaryChannel\} \\
InterChannel_1 &\rightarrow \{InterChannel_1, DistributaryChannel\} \\
DistributaryChannel &\rightarrow \{DistributaryChannel, InterChannel_2, Lobe\} \\
InterChannel_2 &\rightarrow \{InterChannel_2, Lobe\} \\
Lobe &\rightarrow \{Lobe, Lobefringe\} \\
LobeFringe &\rightarrow \{LobeFringe, BasinPlain\} \\
BasinPlain &\rightarrow \{BasinPlain\}
\end{aligned}
$$

Figure 3.6: Orderings of depositional environments

Part II

Underdetermined World Concretization

Chapter 4

Problem Formalization

In this chapter we formally define some concepts that are used to capture the problem domain of the thesis work. The purpose of giving these definitions is to lay the foundation for different methods of representing and solving the problem.

First we define objects which will denote actual objects such as faults and geological units, and where the underdetermination of the objects is captured using variables. Then we define worlds which are used to denote a set of objects and their relations to each other. After that, we show object dependencies which restricts the values a variable of an object can have, meaning the legal instantiations of an underdetermined world are also restricted. Finally, we define the world concretization problem, where the goal is to find all instantiations of an underdetermined world.

4.1 Objects

A variable x is a placeholder for values a. Let $x \leftarrow a$ denote that the variable x is assigned the value a. A value set is a set of values a variable can be assigned.

Definition 1. *Given a variable x, its* value set *V is a set $\{a_1, ..., a_n\}$, where a_i are the only values that can legally be assigned to x.*

For example, the value set of the variable $sealingCapacity$ is $\{Sealing, NonSealing\}$ and the value set of $location$ is $\{FeederChannel, InterChannel_1, DistributaryChannel, InterChannel_2, Lobe, LobeFringe, BasinPlain\}$.

Let $x \Leftarrow V$ denote that V is a value set of variable x. The act of assigning a value to a variable is called a *value assignment*.

Definition 2. *An object o is a pair (id, A), where id is an identifier of the object and A is the set of attributes of the object. Formally, id is a value. Further, A is a set*

$\{a_1, ..., a_m, x_1, ..., x_n\}$, *where* a_i *are attributes with values assigned already and* x_i *are attributes without values assigned yet (variables). Further,* $x_i \Leftarrow V_i$*, where* V_i *is the value set of* x_i*. We say that the attributes* a_i *are* concrete attributes *and* x_i *are* inconcrete attributes*, or* **underdetermined attributes**.

The value of an object's *id* is used to refer to the object. For example the fault object f_1 with the attribute $sealingCapacity$ is written:

$$(f_1, \{sealingCapacity\}),$$

where

$$sealingCapacity \Leftarrow \{Sealing, NonSealing\}.$$

To concretize an object means to assign values to all of its underdetermined attributes.

Definition 3. *A concretized object o is an object* (id, A) *with only concrete attributes in A.*

For example $(f_1, \{Sealing\})$ is a concretized object, it is a concretization of the object f_1. The only other concretization of f_1 is $(f_1, \{NonSealing\})$. The number of concretizations of an object depends on its number of underdetermined attributes and the number of values in the underdetermined attributes' value sets.

4.2 Worlds

A world is a gathering of objects and relations between them.

Definition 4. *A* world *W is a pair* (O, R)*, where O is a set of objects* o_i*, and R is a set of zero or more binary relations between the objects* $r(id_j, id_k)$*. Each object in the world must have a unique identifying value id.*

For example, the three fault objects f_1, f_2 and f_3 can be put into the world $Faults$ as follows

$$Faults = (\{Fault_1, Fault_2, Fault_3\}, \{\}),$$

where $Fault_i = (f_i, \{sealingCapacity_i\})$. A world with only concretized objects is called a *concrete world*, while a world which is not concretized is called an *inconcrete world* or an **underdetermined world**. One legal concretization of $Faults$ is the following concrete world.

$$(\{(f_1, \{Sealing\}), (f_2, \{Sealing\}), (f_3, \{Sealing\})\}, \{\})$$

Depending on the expressiveness needed to represent a proto-scenario, a concretized world either represents a fragment of a proto-scenario or a complete proto-scenario.

We represent the concretization of a world as a tree structure where the root represents the world's initial objects. Each edge represents an assignment of a value to an underdetermined attribute in one of the world's objects. The deeper one goes down in the tree, the closer one gets to a concretized world. Note that the state of the world is underdetermined until reaching the leaves of the tree. We call this type of tree a *value assignment tree*. In figure 4.1 we show the value assignment tree of the world *Faults*. In this example S_i means the variable $sealingCapacity_i$ is assigned $Sealing$, and $\overline{S_i}$ means it is assigned $NonSealing$.

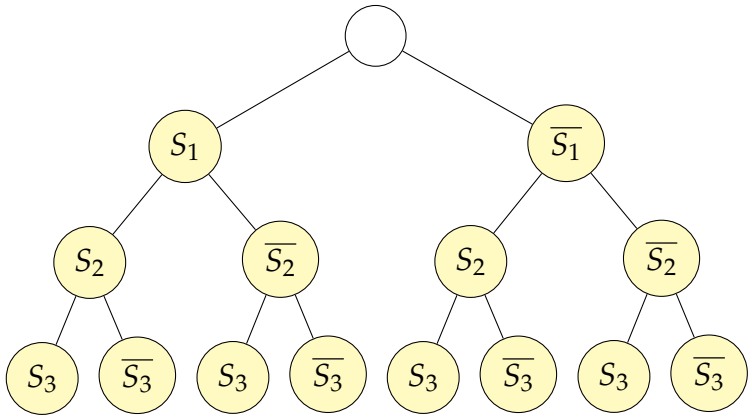


Figure 4.1: Value assignment tree of the *Faults* world

Note that a value assignment tree is not necessarily a binary tree. The tree shown in Figure 4.1 is binary because the value set of the underdetermined attributes of the objects in the world contains only two values.

4.3 Object Dependencies

Sometimes the legality of a value assignment of a variable in an object is dependent on the value of another variable in the world the object is in. It can be a variable in the same object or variables in other objects. These dependencies lead to some value assignments being illegal, restricting the number of concrete worlds.

Definition 5. *An object dependency is when the value of an attribute in an object depends on the value of another attribute in the world.*

Assume we have three geological unit objects gu_5, gu_8 and gu_{11} with a variable for environment location: $(gu_i, \{location_i\})$, where $location_i \Leftarrow \{Lobe, LobeFringe, BasinPlain\}$. The dependency of a geological unit's location in parts of the submarine fan can be denoted as follows.

$$\begin{aligned} Lobe &\rightarrow \{Lobe, LobeFringe\} \\ LobeFringe &\rightarrow \{LobeFringe, BasinPlain\} \\ BasinPlain &\rightarrow \{BasinPlain\} \end{aligned}$$

What is expressed is that *Lobe* is restricted to only be in front of another *Lobe* or a *LobeFringe*, and similarly *LobeFringe* is restricted to only be in front of another *LobeFringe* or a *BasinPlain*. Furthermore, a *BasinPlain* is restricted to only be in front of another *BasinPlain*.

We define this binary relation between two geological unit objects as $frontOf(gu_i, gu_j)$, which represents that the geological unit object with id gu_i is directly in front of the geological unit object with id gu_j. This restriction creates dependencies in the value assignments of objects during concretizations. More specifically, it creates dependencies between the objects, *object dependencies*. For example, if $frontOf(gu_5, gu_8) \in R$, and the object with identity gu_5 has *BasinPlain* assigned to its *location* attribute, it is not allowed for gu_8 to be assigned *Lobe* as the value for its *location* attribute.

To show the effects of object dependencies, we present the world $Rocks_R$ which consists of three objects with one location attribute each:

$$Rocks_R = (O_R, R_R),$$

where

$$O_R = \{(gu_5, \{location_5\}), (gu_8, \{location_8\}), (gu_{11}, \{location_{11}\})\},$$

and

$$R_R = \{frontOf(gu_5, gu_8), frontOf(gu_8, gu_{11})\},$$

and

$$location_i \Leftarrow \{Lobe, LobeFringe, BasinPlain\}.$$

Figure 4.2 shows a value assignment tree for the $Rocks_R$ world. In the tree, LB_i, LF_i and BP_i means $location_i$ was assigned *Lobe*, *LobeFringe* and *BasinPlain*, respectively. The yellow nodes represent legal value assignments of the variables, and thus a branch with only yellow nodes is a concretization of $Rocks_R$. Any branch with a red node is illegal, and as we can see in the figure, for $Rocks_R$ 19 of the 27 possible concretizations based on the value sets are illegal due to the dependencies.

To show the effects of a partial concretization, we create a partly concretized world of $Rocks_R$, which we call $Rocks_P$. The difference is the assignment of *lobe* to the *location* variable of gu_5.

$$Rocks_P = (O_P, R_P),$$

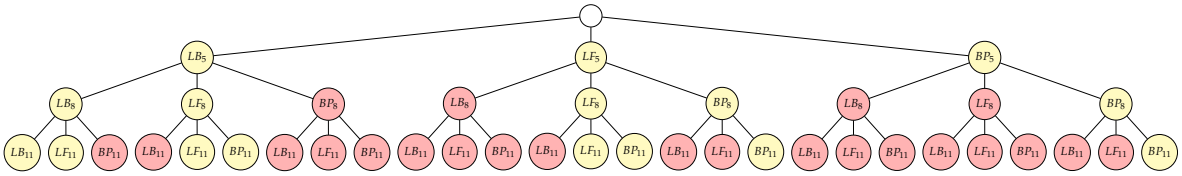


Figure 4.2: Value assignment tree of world $Rocks_R$

where

$$O_P = \{(gu_5, \{\textbf{\textit{Lobe}}\}), (gu_8, \{location_8\}), (gu_{11}, \{location_{11}\})\},$$

and

$$R_P = \{frontOf(gu_5, gu_8), frontOf(gu_8, gu_{11})\},$$

and

$$location_i \Leftarrow \{Lobe, LobeFringe, BasinPlain\}.$$

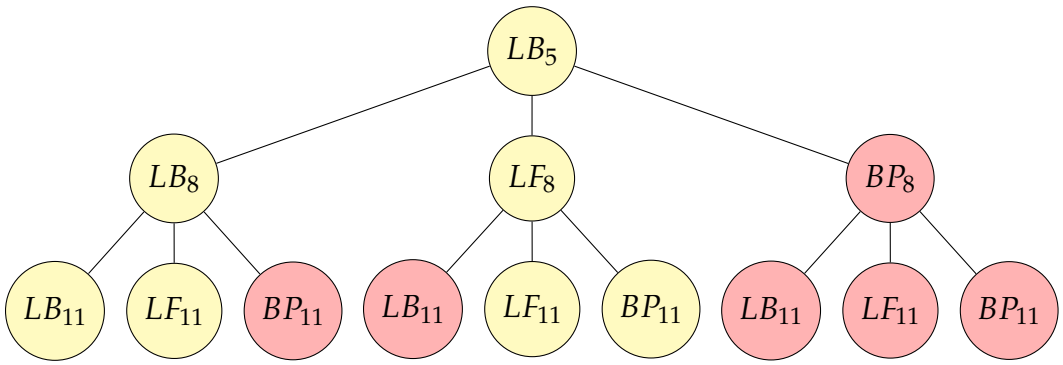


Figure 4.3: Value assignment tree of world $Rocks_P$

Figure 4.3 shows a value assignment tree for the $Rocks_P$ world. The root of the tree is marked LB_5 as the object gu_5 is already concretized, and its *location* variable is set to *Lobe*. If one looks closely at both of the trees, one can see that Figure 4.3 is the left branch of the root of the tree in Figure 4.2. The size of $Rocks_P$ is one third of $Rocks_R$, meaning that two thirds of the search space was removed by concretizing an attribute. In fact, the value assignment tree has an exponential growth as the concretization is a combinatorial problem.

4.4 World Concretization

The main problem we work on in this thesis is the act of finding all concrete worlds, given an underdetermined world and possibly restrictions on the value assignments,

which we call the *world concretization problem*.

Definition 6. *The world concretization problem is to find all concrete world instantiations of an underdetermined world based on the given domain knowledge.*

In particular, we try to concretize a world representing the model of the use case, which we here represent as the world *UseCase*.

$$UseCase = (O_U, R_U)$$

The objects of the world are the faults f_0, f_1, f_2 and f_3, and the geological units gu_5, gu_8, gu_{11}, gu_{14}. O_U is defined as follows.

$$O_U = \{(f_0, \{sealingCapacity_0\}), (f_1, \{sealingCapacity_1\}), (f_2, \{sealingCapacity_2\}),$$

$$(f_3, \{sealingCapacity_3\}),$$

$$(gu_5, \{location_5, permeability_5, porosity_5\}),$$

$$(gu_8, \{location_8, permeability_8, porosity_8\}),$$

$$(gu_{11}, \{location_{11}, permeability_{11}, porosity_{11}\}),$$

$$(gu_{14}, \{location_{14}, permeability_{14}, porosity_{14}\})\}$$

Each of the objects' attributes have the following value sets.

$$sealingCapacity_i \Leftarrow \{Sealing, NonSealing\},$$

$$permeability_i \Leftarrow \{Permeable, NonPermeable\},$$

$$porosity_i \Leftarrow \{Porous, NonPorous\},$$

and

$$location_i \Leftarrow \{FeederChannel, InterChannel_1, DistributaryChannel,$$

$$InterChannel_2, Lobe, LobeFringe, BasinPlain\}.$$

The geological units of the world is in a specific order, formalized with the $frontOf$ relation. R_U is defined as follows.

$$R_U = \{frontOf(gu_5, gu_8), frontOf(gu_8, gu_{11}), frontOf(gu_{11}, gu_{14})\}$$

location is object dependent through the $frontOf$ relation as in the description of logical distribution of components in the submarine fan in Section 3.4 on page 33.

4.5 Summary

To sum up, in this chapter we have formalized the world concept and the different entities within it. In particular, we have formalized the terms underdetermined world and world concretization.

In the next chapter we present an approach to represent underdetermined worlds with OWL ontologies, and an algorithm using the OWL 2 API to concretize these types of ontologies. In Section 6.1 on page 66 we model the *UseCase* world with DL.

Chapter 5

Concretizing Worlds with OWL Ontologies

In this chapter we introduce a methodology which uses OWL 2[1] [11] ontologies to generate the concretized worlds of an underdetermined world. For the rest of the text we refer to OWL 2 ontologies as OWL ontologies. We use OWL because it is a language where one can formalize domain knowledge with description logic, and it has open source tool support such as reasoners and the OWL API[2] [14]. We develop an approach for solving the world concretization problem where domain knowledge is stored in an OWL ontology, and the world concretization computation is done by an algorithm connected to the ontology through the OWL API. In particular, we make use of the class hierarchy and object properties to model the domain and its dependencies, the OWL world instance to represent a world, and consistency checking to verify the legalities of value assignments.

5.1 Ontologies and Worlds

Before we present the algorithm, we show a methodology for modeling the worlds we defined in Section 4.2 with OWL ontologies. We use the ABox to represent a specific world, while the TBox and RBox are used to formalize the domain knowledge of the world. The methodology is designed such that a consistent ontology with an ABox where no more class assertions can be added represents a concrete world, while other ABoxes represent underdetermined worlds.

Figure 5.1 shows the mapping of the entities of a world to entities of an OWL ontology. Object IDs are represented in OWL with individual names, and

[1] `https://www.w3.org/OWL/`

[2] Source code and wiki at `https://github.com/owlcs/owlapi`

World		OWL
Object ID (*id*)	→	Individual Name
Binary Relations	→	Object/Data Properties
Value Name	→	Value Class
Value Set Name	→	Value Set Class
Object Name	→	Object Class

Figure 5.1: World entities mapped to OWL entities

binary relations are represented with properties. Values, value sets and objects are represented with classes. In the next section, we specify how these classes are formalized in a TBox to make all consistent ABoxes represent worlds with only objects. We do not specify a methodology for formalizing object dependencies as these will differ with the knowledge being formalized.

5.1.1 Modeling Objects with OWL

In this section we show how objects can be modeled with OWL, while the next section contains an example of a world represented with OWL. An object is defined in Chapter 4.2 as a pair with an identifier and a set of attributes, where some of the attributes are variables with value sets. We now design the modeling methodology that uses OWL classes to represent objects, attributes and value sets. The main goal of the methodology is to make sure only the objects of a world that are of type T can be in the class T. For example, for an individual gu_5 to be a geological unit, it must have the attributes *location*, *permeability* and *porosity*, and it cannot have the *sealingCapacity* attribute.

To represent an object attribute we use a class that represents a value set. Each value in a value set is represented with a (value) class, and the value set class is the disjoint union of its value classes. A value set class V is modeled as follows:

$$V \equiv Z_1 \sqcup Z_2 \sqcup \ ... \ \sqcup Z_n,$$

where Z_i are V's value classes and n is the number of values in the value set of the attribute represented. The value classes are made disjoint by the following axioms.

$$Z_1 \sqcap Z_2 \sqsubseteq \bot,\ Z_1 \sqcap Z_3 \sqsubseteq \bot,\ ...,\ Z_{n-1} \sqcap Z_n \sqsubseteq \bot$$

An object class represents the set of all object IDs in a world which is of that object type. Thus we model the object class as an intersection of all of its value set classes:

$$O \equiv \prod_{i=1}^{m} V_i$$

where V_i are value set classes and m is the number of attributes the objects in O have. Further, an object class is disjoint with value set classes that do not belong to any of its attribute variables:

$$O \sqsubseteq \bigsqcup_{i=1}^{k} \neg W_i$$

where W_i are value set objects, k is the number of attributes in the world the objects in O do not have and there are no variables x in the objects in O such that $x \Leftarrow W_i$.

By modeling an object like this, the OWL reasoner makes sure that an ontology with any ABox in which there are variables that cannot be concretized is inconsistent. Further, if several values in a value set can be assigned to a variable at once, it is also an inconsistent case. This is due to the disjoint union. In the next section we use the methodology shown here to model faults.

5.1.2 An Example World in OWL

We model the *Fault* object with the value set class *SealingCapacity* in the TBox $\mathcal{T}_F$ as shown in Figure 5.2.

$$\begin{array}{c} Fault \equiv SealingCapacity \\ SealingCapacity \equiv Sealing \sqcup NonSealing \\ Sealing \sqcap NonSealing \sqsubseteq \bot \end{array}$$

Figure 5.2: Fault object TBox $\mathcal{T}_F$

With $\mathcal{T}_F$, we can model an underdetermined world consisting of one fault with the ABox $\mathcal{A}_F$ that only has the class assertion $Fault(f_1)$:

$$\{Fault(f_1)\}$$

$\mathcal{T}_F$ and $\mathcal{A}_F$ in combination represent the world

$$(\{(f_1, \{sealingCapacity\})\}, \{\}),$$

where

$$sealingCapacity \Leftarrow \{Sealing, NonSealing\}.$$

The concretizations of this world are represented by ABoxes $\mathcal{A}_{F_1}$ and $\mathcal{A}_{F_2}$.

$$\mathcal{A}_{F_1} = \{Fault(f_1), SealingCapacity(f_1), Sealing(f_1)\}$$
$$\mathcal{A}_{F_2} = \{Fault(f_1), SealingCapacity(f_1), NonSealing(f_1)\}$$

In Figure 5.3 we show the results of a concretization using a simplified version of $\mathcal{T}_F$, i.e. $\mathcal{T}_S$, and an ABox $\mathcal{A}_S$ representing an underdetermined world with two faults. $\mathcal{T}_S$ is simplified by removing the value set class *SealingCapacity*, and this is done to make the ABoxes in the example smaller. If $\mathcal{T}_F$ was used instead, the only difference would be that each ABox would also have a *SealingCapacity* class assertion for each individual. The Underdetermined Ontology Concretizer in the figure represents a procedure which generates all of the consistent maximal ABoxes of an ontology.

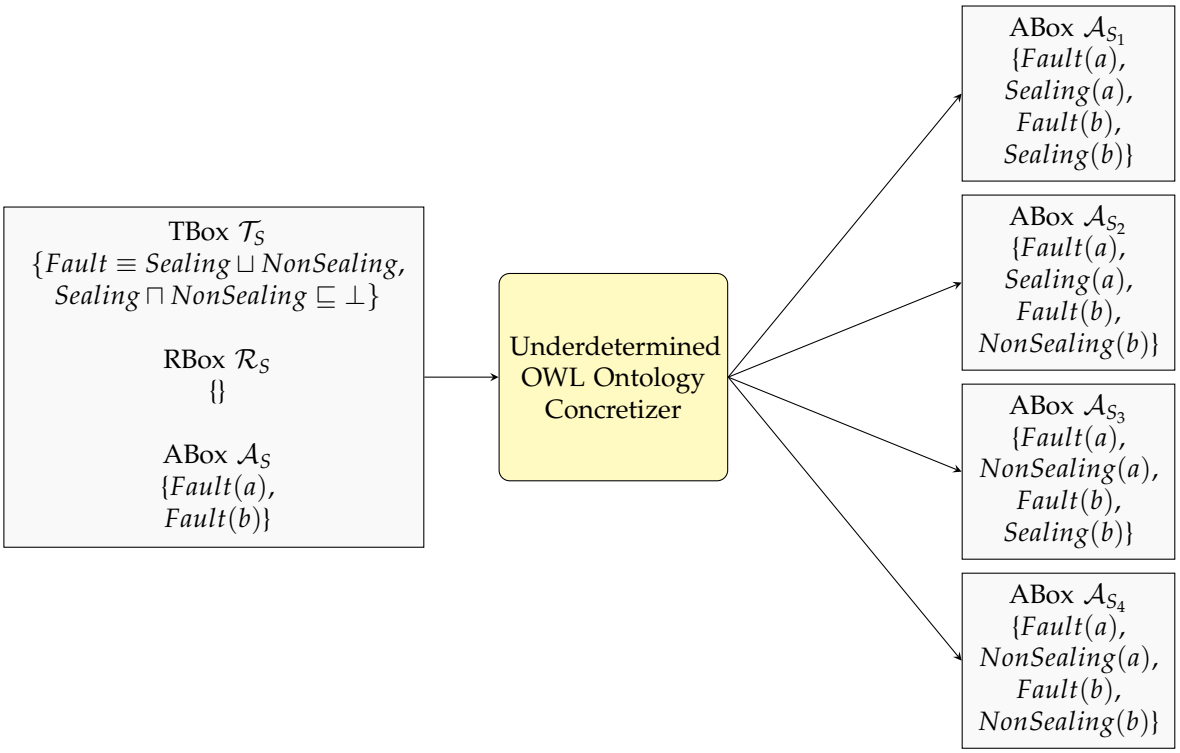


Figure 5.3: Underdetermined ontology concretization

5.2 Consistent Maximal ABox Algorithm

In this section we present an algorithm which finds all consistent maximal ABoxes of an ontology. With maximal, we mean that no class assertions can be added to the ABox while the ontology remain consistent. Because OWL is used, we use OWL terminology instead of DL terminology as presented in 2.1.5. First we present a naive algorithm, and then an optimized algorithm utilizing properties of OWL ontologies is presented.

5.2.1 Naive Algorithm

The naive algorithm generates the power set of all possible class assertions of an ontology given the set of classes C and the set of individuals I in the ontology. A class assertion is of the form $A(i)$ where A is a class and i is an individual. To begin with, it generates the set of all class assertions possible given the classes and individuals. For example, an ontology where $C = \{A, B\}$ and $I = \{j, k\}$ has the following sets of class assertions:

$$\{A(j), B(j)\}$$

and

$$\{A(k), B(k)\}.$$

These are combined into a set of class assertions for all of the individuals C_A as

$$C_A = \{A(j), B(j)\} \cup \{A(k), B(k)\}.$$

The naive algorithm computes the power set of C_A, $\mathcal{P}(C_A)$. We call this power set the *power box* of the ontology. When giving C and I as inputs to Algorithm 1, it will return the following set of 16 combinations.

$$\{\{A(j), B(j), A(k), B(k)\}, \{A(j), B(j), A(k)\}, \{A(j), B(j), B(k)\},$$

$$\{A(j), B(j)\}, \{A(j), A(k), B(k)\}, \{A(j), A(k)\}, \{A(j), B(k)\}, \{A(j)\},$$

$$\{B(j), A(k), B(k)\}, \{B(j), A(k)\}, \{B(j), B(k)\},$$

$$\{B(j)\}, \{A(k), B(k)\}, \{A(k)\}, \{B(k)\}, \{\}\}$$

To only get the legal concretized worlds, most of the sets in $\mathcal{P}(C_A)$ must be discarded after the algorithm is run. Specifically, each set which is inconsistent must be discarded because they represent worlds with illegal assignments. Further, any set which is a subset of another consistent set must be discarded as they are underdetermined. The result after discarding these sets is a set of ABoxes representing all of the concrete worlds. For example, if the TBox of the ontology states that the classes A and B are disjoint, then the combination $\{A(j), B(j)\}$ would have to be removed because of inconsistency. The following 4 combinations would be consistent and maximal.

$$\{A(j), A(k)\}, \{A(j), B(k)\}, \{B(j), A(k)\}, \{B(j), B(k)\}$$

Any ontology which is a subset of any of the 4 combinations above would have to be removed because it is not maximal and therefore underdetermined, for example $\{A(j)\}$.

Algorithm 1 PowerBox Algorithm

Preconditions: The set of individuals from the ABox of the ontology in the variable *individuals* and the set of classes of the class hierarchy in the variable *classes*.
Postcondition: All unique combinations of the elements in *assertions* stored in *powerBox*.

```
 1: procedure INIT(individuals, classes)
 2:     global powerBox ← new set            ▷ variable reachable from all procedures
 3:     C_A ← ALGORITHMSET(individuals, classes)
 4:     abox ← new set
 5:     POWERBOX(C_A, abox)
 6:     return powerBox
 7:
 8: procedure ALGORITHMSET(individuals, classes)
 9:     C_A ← new set
10:     for all i ∈ individuals do
11:         for all c ∈ classes do
12:             axiom ← c(i)                 ▷ individual i asserted to be of class c
13:             C_A.add(axiom)
14:     return C_A
15:
16: procedure POWERBOX(assertions, abox_1)
17:     if |assertions| = 0 then
18:         powerBox.add(abox_1)
19:         return
20:
21:     axiom ← assertions.pop()
22:     abox_2 ← abox_1.copy().add(axiom)
23:     POWERBOX(assertions.copy(), abox_2)
24:
25:     POWERBOX(assertions.copy(), abox_1)
```

Algorithm Description

In Algorithm 1, INIT is the procedure which initializes the computation. It has two parameters: Let *individuals* be the list of individuals (I) and *classes* be a list which represents the set of classes (C). The global variable *powerBox* is where $\mathcal{P}(C_A)$ is stored, and it is generated by the POWERBOX procedure. The set C_A is generated by the ALGORITHMSET procedure. In the pseudo code, the symbol $\triangleright$ is used for comments. After Algorithm 1 is run, one can find the concretized worlds by discarding any set which is a subset of another set in *powerBox* and discarding all sets for which the ontology is inconsistent. This discarding is not a part of the depicted algorithm. In Section 5.2.3 below, we propose an optimized algorithm which discards the inconsistent ones by using the OWL reasoner to check the legalities of each value assignment.

5.2.2 Reasoner Module

Before we present the optimized algorithm, we should note a design choice of using the reasoner as a module. The algorithm generates value assignments, and the reasoner is used to check the legalities of each value assignment with consistency checks.

We could have chosen to modify a proof search algorithm to generate the same results. The comparative benefit would be removing a layer of traversal through the class hierarchy, possibly having a better complexity and likely being more efficient during computations. However, this would mean a less modular solution where we would have been restricted to using the search algorithm we created.

So we decided to use the reasoner as a module, with one benefit being the ability to quickly swap between already developed reasoners. This could be useful when using the algorithm on different ontologies. Some reasoners are optimized for a specific fragment of DL, such as ontop being optimized for OWL 2 QL [19] and RDFox being optimized for OWL 2 RL [28].[3] Another benefit is that the algorithm itself is cleaner as the technical details of the reasoner is abstracted away. Furthermore, the algorithm proposed, or a modification of it, might have other uses than what it is applied for in this thesis. If we were to make a reasoner specifically for our use, other applications would be less likely due to the algorithm being more specialized.

[3]See [25] for a review of reasoners and e.g. `http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/` for a list of reasoners.

5.2.3 Optimization of the Naive Algorithm

In an OWL ontology it is not possible to add any assertions to an inconsistent ontology to make it consistent. We utilize this property to optimize the naive algorithm by discarding inconsistent ABoxes during computation. Thus not continuing value assignments in branches that are known to be illegal, and as a result decreasing the search space of the generation of class assertion combinations.

Algorithm Description

The optimized Algorithm 2 (the algorithm)[4] assumes access to the OWL API with an instantiated OWLOntology object *ont* and through that access to a reasoner and all of the entities in the ontology such as the class hierarchy. *ont* is the initial OWL ontology input to the algorithm. In the pseudo code, the reasoner is named *reasoner*, and the class hierarchy is accessed through the root $\top$ and its sub classes. Further, it assumes access to an empty set of class assertion combinations called *combs*. The parameter *individuals* is the set of named individuals in the ontology. The goal of the algorithm is to compute and store all consistent maximal ABoxes of the given ontology.

The algorithm uses a FIFO queue of classes to ensure breadth-first traversal of the class hierarchy. The queue is updated through the procedure QUEUEUPDATER, where the direct sub classes of a class is added to the end of the queue.

The algorithm starts in NEXTINDIVIDUAL with *individuals*. The first individual is popped from the set, and then the algorithm traverses the class hierarchy with the individual to create combinations of class assertions with the individual by calling TREETRAVERSE. The traversal of the class hierarchy is done such that the algorithm will continue first with (line 29) and then without (line 34) a class assertion. By continuing first with the class assertion, it is ensured that when each combination reaches the base case, they are not subset of any combination that reaches the base case at a later stage of the algorithm. This property of the algorithm is used to select which combinations to store. If a class assertion makes the ontology inconsistent, the algorithm will only continue without the assertion.

To make sure the semantics of the ontology is kept intact at each step of the algorithm, an assertion axiom is only added to (line 23), and subsequently removed from (line 32), the ontology if it was not a part of the initial ontology. When an axiom is added to the ontology, the reasoner is synchronized. To synchronize the reasoner means to make the reasoner handle the axioms added or removed since it was last synchronized. Note that the reasoner does not need to compute all of the previously generated entailments when synchronizing.

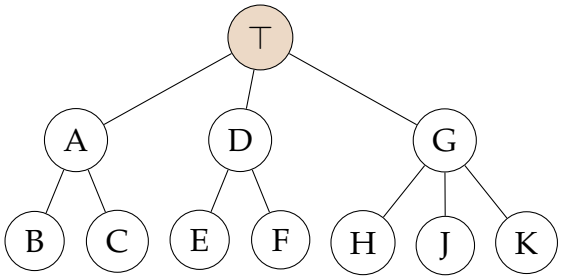
[4] See `https://github.com/vskaret/consistent-maximal-abox-generator` for our implementation

Algorithm 2 Consistent Maximal ABox Generator

Preconditions: A consistent OWL ontology *ont* with a non-empty set of named individuals in its ABox making up the *individuals* parameter, an empty set *comb* and access to a global empty set *combs*.

Postconditions: All of, and only, the consistent maximal combinations in *combs*.

```
1:  procedure NEXTINDIVIDUAL(individuals, comb)
2:      individuals_1 ← individuals.copy()
3:      u ← individuals_1.remove(0)
4:      queue ← QUEUEUPDATER(empty queue, ⊤)
5:      TREETRAVERSE(individuals_1, queue, u, comb)
6:  procedure QUEUEUPDATER(queue, class)
7:      temp ← queue.copy()
8:      for all (sub ⊑ class), where sub is a direct subclass of class do
9:          add sub to temp
10:     return temp
11: procedure TREETRAVERSE(individuals, queue_1, u, comb_1)
12:     if |queue_1| = 0 then
13:         if |individuals| > 0 then
14:             NEXTINDIVIDUAL(individuals, comb_1)              ▷ next individual
15:         else if ¬(comb_1 ⊆ c), where c ∈ combs then
16:             combs.add(comb_1)                                ▷ combination found
17:         return
18:     class ← pop queue_1
19:     axiom ← assert class(u)
20:     ontologyContainedAxiom ← ontology.contains(axiom)
21:
22:     if ¬ontologyContainedAxiom then
23:         ontology.add(axiom)
24:         synchronize reasoner
25:
26:     if ontology is consistent then
27:         comb_2 ← comb_1.copy().add(axiom)
28:         queue_2 ← QUEUEUPDATER(queue_1, class)
29:         TREETRAVERSE(individuals, queue_2, u, comb_2)
30:
31:     if ¬ontologyContainedAxiom then
32:         ontology.remove(axiom)
33:         synchronize reasoner
34:     TREETRAVERSE(individuals, queue_1, u, comb_1)
```



Disjointness Axioms

- $A \sqcap D \sqsubseteq \bot$
- $A \sqcap J \sqsubseteq \bot$
- $A \sqcap K \sqsubseteq \bot$
- $B \sqcap C \sqsubseteq \bot$
- $D \sqcap H \sqsubseteq \bot$
- $D \sqcap K \sqsubseteq \bot$
- $H \sqcap J \sqsubseteq \bot$
- $H \sqcap K \sqsubseteq \bot$
- $J \sqcap K \sqsubseteq \bot$

Figure 5.4: Example class hierarchy

When the traversal is finished with one set of class assertions for the individual, satisfying the if condition on line 12, it will either continue with the next individual or attempt to store the combination. By continuing with the next individual for each combination of the previous individual(s), the algorithm ensures that all sets of class assertion combinations including all individuals are generated. By storing only combinations that are not subsets of any combinations already stored, it ensures that only maximal combinations are stored.

5.2.4 Example Algorithm Run

To visualize how the algorithm works, we show an example of the traversal of the class hierarchy with one individual. The graph in figure 5.4 depicts the class hierarchy with its disjointness axioms to the right. The algorithm works with other axioms as well, but we find it best to show how it works with an example using only disjointness axioms. The example run from Figure 5.5 to Figure 5.19 visualizes the traversal and the base cases where the consistent maximal ABoxes are found. The example is quite long because it shows step by step how the algorithm works and brings forth an edge case where a class is not combinable with most other classes.

The figures in the example run of the algorithm use the following node coloring scheme, which symbolizes the different elements of the algorithm:

Class not yet reached

Root of the class hierarchy

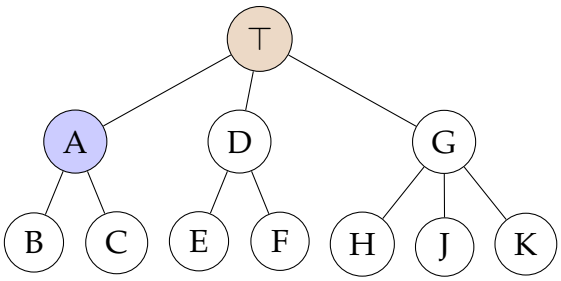
Current class assertion

Ontology consistent after assertion

Ontology inconsistent after previous assertion

Class skipped for the individual in the current combination computation

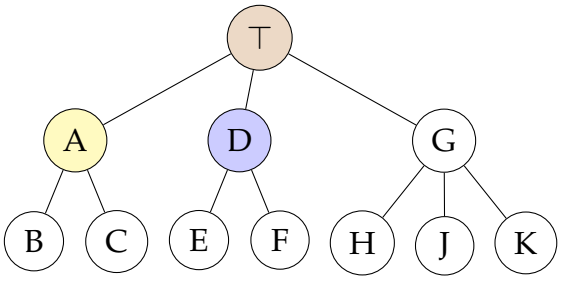
The figures are split with the class hierarchy on the left, and a list of class assertion combinations on the right. The list with the highest number is the one currently being computed, and the steps of the algorithm are described in the captions of the figures. Three vertical dots ⋮ between figures represents skipping of steps.



Instantiations

1. A(i)

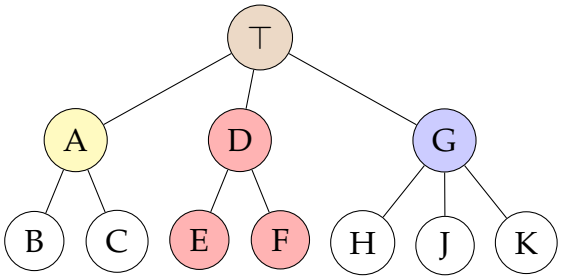
Figure 5.5: Add A(i)



Instantiations

1. A(i), D(i)

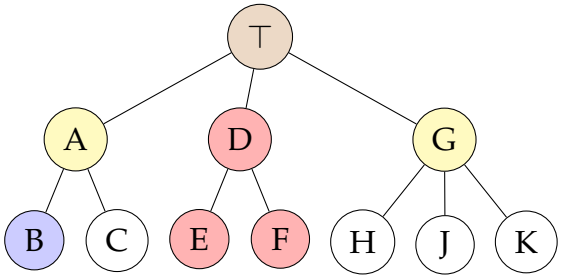
Figure 5.6: Consistent. Add D(i)



Instantiations

1. A(i), G(i)

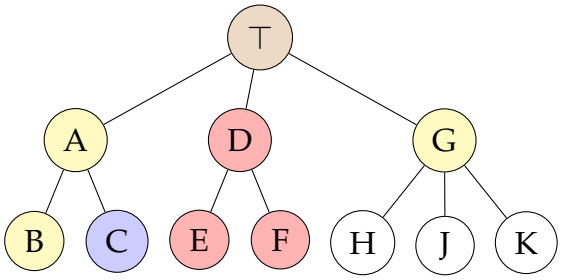
Figure 5.7: Inconsistent due to axiom $A \sqcap D \sqsubseteq \bot$.
Continue without D(i) and add G(i)



Instantiations

1. A(i), G(i), B(i)

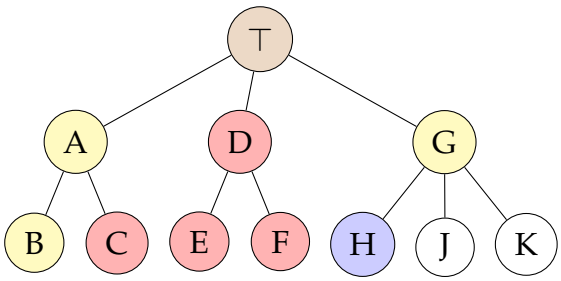
Figure 5.8: Consistent. Add B(i)



Instantiations

1. A(i), G(i), B(i), C(i)

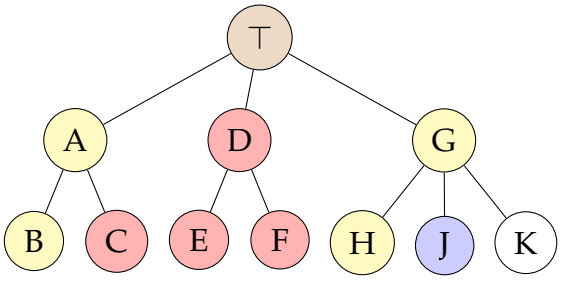
Figure 5.9: Consistent. Add C(i)



Instantiations

1. A(i), G(i), B(i), H(i)

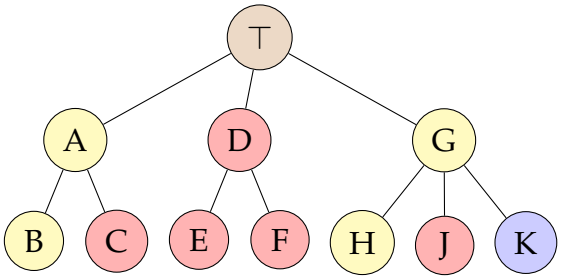
Figure 5.10: Inconsistent due to axiom $B \sqcap C \sqsubseteq \bot$.
Continue without C(i) and add H(i)



Instantiations

1. A(i), G(i), B(i), H(i), J(i)

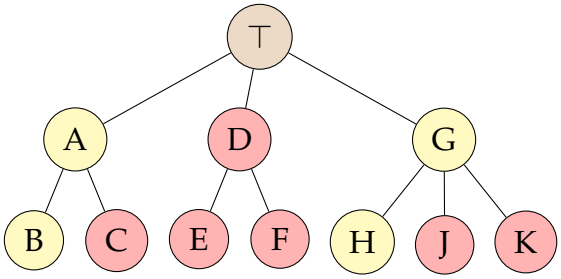
Figure 5.11: Consistent. Add J(i)



Instantiations

1. A(i), G(i), B(i), H(i), K(i)

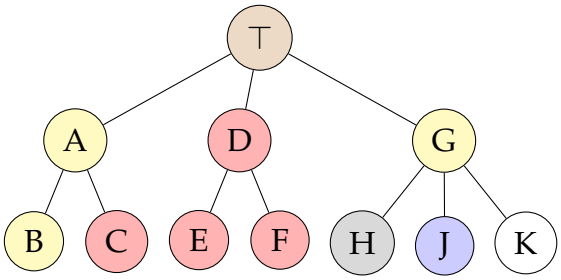
Figure 5.12: Inconsistent due to axiom $H \sqcap J \sqsubseteq \bot$. Continue without J(i) and add K(i)



Instantiations

1. A(i), G(i), B(i), H(i)

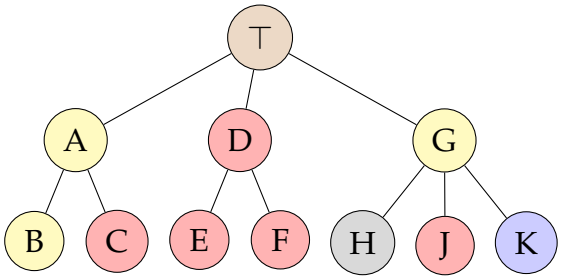
Figure 5.13: Inconsistent due to axiom $H \sqcap K \sqsubseteq \bot$. Remove K(i). **End of graph, one combination discovered**



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), B(i), J(i)

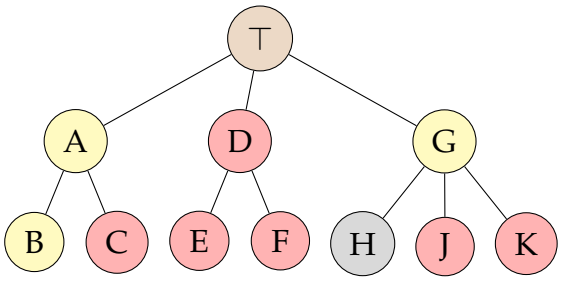
Figure 5.14: Backtrack and attempt without H(i)



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), B(i), K(i)

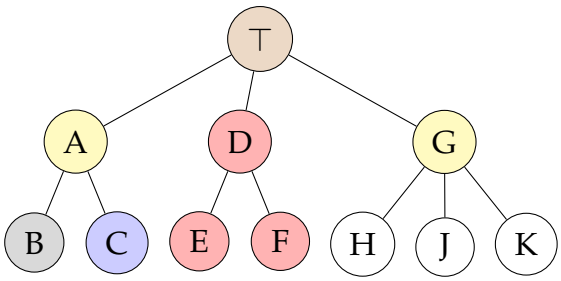
Figure 5.15: Inconsistent due to axiom $A \sqcap J \sqsubseteq \bot$. Continue without J(i) and add K(i)



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), B(i)

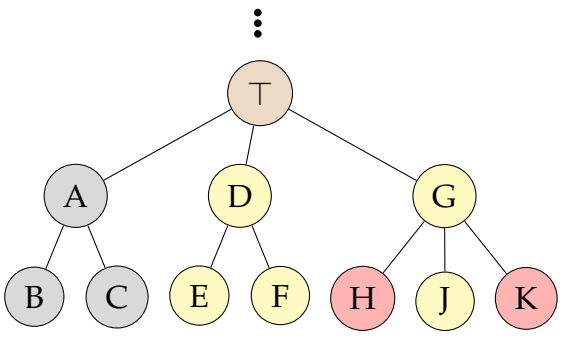
Figure 5.16: Inconsistent. Remove K(i). **End of graph, but combination not stored because it is a subset of another combination**



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), C(i)

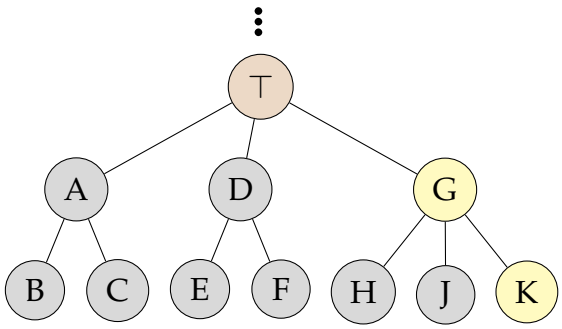
Figure 5.17: Backtrack and attempt without B.



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), C(i), H(i)
3. D(i), G(i), E(i), F(i), J(i)

Figure 5.18: Combination which starts with D(i). Many steps skipped since previous figure



Instantiations

1. A(i), G(i), B(i), H(i)
2. A(i), G(i), C(i), H(i)
3. D(i), G(i), E(i), F(i), J(i)
4. G(i), K(i)

Figure 5.19: Last combination of the example

The figures of the example show how the consistent maximal combinations are found. In the next section, we prove that the algorithm generates sound and complete results given some assumptions about the algorithm.

5.2.5 Sound and Complete Results

For Algorithm 2 (the algorithm) to be useful, it must do what it is supposed to do. In this section we prove that the results generated by the algorithm are correct assuming the algorithm generates all combinations of consistent combinations of class assertions and that the QUEUEUPDATER procedure correctly updates the queue for breadth-first traversal. **The results generated are combinations of class assertions, which we for briefness call combinations.**

To be correct, the results must be sound and complete. Sound in the sense that only combinations that are legal and maximal are stored. Complete in the sense that all of the legal and maximal combinations are found and stored. A legal combination is one which is a part of a consistent ontology. A maximal combination is one which is not a subset of any other legal combination given the initial ontology. **For briefness, we say that a combination *c* is consistent when the ABox generated at the same step as *c* in the algorithm is part of a consistent ontology.**

We first argue for three lemmas which we later use to show soundness and completeness. The first is that only legal combinations are stored. The second is that only maximal combinations are stored. The third is that the algorithm stores any legal and maximal combination. These lemmas are later used to show soundness and completeness.

Lemma 1. *All combinations c stored in the variable combs at the end of a run of the algorithm are consistent.*

To show that the lemma holds, we prove by induction that $comb_1$, which is the variable that is stored on line 16, is consistent at each recursive step of the algorithm. If $comb_1$ is consistent at each step of the recursion, any combination stored in the

base case of the algorithm is also consistent. The parameter n in the induction is the step of recursion of the procedure. So the induction step is to prove that if $comb_1$ is consistent at an arbitrary step of the recursion, then the $comb_1$ of the next step will also be consistent.

Base case n = 0: At the first step of the recursion $comb_1$ is consistent as a precondition of the algorithm is that the ontology is consistent and $comb_1$ is empty in the beginning.

Induction step: If $comb_1$ is consistent at step $n = k$, then it is also consistent at step $n = k + 1$. This must be shown to hold for the following two cases.

1. The first is when TREETRAVERSE continues its traversal on (a) line 29 and (b) line 34.

2. The other case is when TREETRAVERSE reaches its base case and there are still individuals left on line 14.

Case 1 (a): When TREETRAVERSE continues its traversal on line 29, it continues with $comb_2$, which is a copy of $comb_1$ with another class assertion axiom added to it. The axiom is created on line 19. On lines 22-24 the axiom is added to the ABox of the ontology (if it was not already a part of it), and the reasoner is synchronized.

On lines 26-29, if the ontology is consistent with the new axiom added the following happens, $comb_2$ is created as a copy of $comb_1$ including the new axiom that was added to the ontology, and $queue_2$ is created as the queue for the breadth-first search order from the current node in the class hierarchy. Lastly, the traversal for the current individual is continued with $comb_2$ and $queue_2$. As $comb_2$ corresponds to an ABox which is part of a consistent ontology, $comb_2$ is consistent at line 29 and thus $comb_1$ in the next step is consistent.

Case 1 (b): When TREETRAVERSE continues its traversal on line 34, it continues with $comb_1$, which is consistent at the current step and not modified at all. Thus $comb_1$ is also consistent at the next step. Furthermore, the ontology is the same as at the start of the recursive step. If the axiom created at this step was not a part of the ontology, it is first added to the ontology on lines 22-24 and then removed on lines 31-33. If it was a part of the ontology, the lines 22-24 and 31-33 which modifies the ontology are never executed.

Case 2: When the algorithm reaches line 14, nothing changes with $comb_1$ before the next step. Only a new individual u and the breadth-first search queue is updated on lines 1-4. Thus $comb_1$ is also consistent when TREETRAVERSE is called on line 5.

Lemma 2. *At the end of the algorithm, there exists no combinations b and c such that* $b \in combs$, $c \in combs$ *and* $b \subset c$.

To show that the lemma holds, we show first that the combinations are generated in an order such that a combination is never a subset of a combination generated

later. Then we show that at the base case of the algorithm only combinations which are not subset of any of the already stored combinations are stored.

First, assume for contradiction that $b \subset c$, where b and c are combinations and c reaches the base case of the algorithm (on line 15) later than b. Because $b \subset c$, all elements of c are in b. On each step of the algorithm, $comb_2$, if consistent, reaches the base case before $comb_1$. This is because $comb_2$ continues its traversal on line 29, while $comb_1$ continues its traversal on line 34, and there are no ways for line 34 to execute before line 29 (if line 29 is executed). ($comb_2$ will on its next step of the recursion be $comb_1$ and finish the run of only executing line 34 where no elements are added to the combination before the recursion returns to the previous step.) Thus, for the assumption to hold, at some step it must hold that $comb_2 \subset comb_1$. However, $comb_2$ is always a copy of $comb_1$ with another axiom not in $comb_1$ added. Thus $comb_2 \subset comb_1$ and $comb_2 = comb_1 \cup \{a\}$ where a is an arbitrary class assertion axiom not in $comb_1$. This is a contradiction, and therefore a combination is never a subset of a combination that reaches the base case of the algorithm after it.

Second, assume for contradiction that a combination b is stored in the variable *combs* on line 16 when there is already a combination c stored such that $b \subset c$. A combination is only stored on line 16 of the algorithm, and line 16 is only executed if the if statement on line 15 holds. The if statement holds if $\neg(b \subseteq c)$ holds, where c is an arbitrary combination in *combs*. Then there is a contradiction because $\neg(b \subseteq c)$ and $b \subset c$ cannot hold at the same time. Thus the negation of the assumption holds, which is that no such combination b is stored, and therefore the lemma holds.

Lemma 3. *Any consistent combination b, with no consistent combination c such that $b \subset c$, is stored in the variable combs.*

Assume for contradiction that a consistent combination b, where there exists no consistent combination c such that $b \subset c$, is generated by the algorithm and not stored in *combs* . Then the algorithm never reached line 16 with b. That means that the if statement on line 15, in this case $\neg(b \subseteq c)$ where $c \in combs$, was false. Thus, $b \subseteq c$. However, we assumed there was no c such that $b \subset c$ so this is a contradiction. Therefore b is stored and the lemma holds.

Theorem 1 (Soundness)**.** *Only consistent combinations b of class assertions, with no consistent combination c found such that b is a subset of c, are stored.*

This theorem holds due to Lemma 1 (all combinations stored in *combs* are consistent) and Lemma 2 (only combinations which do not have consistent supersets are stored in *combs*).

Theorem 2 (Completeness)**.** *All consistent combinations are generated, and there exists no consistent combination which is a superset of any of the combinations stored in the variable combs.*

All possible consistent combinations of class assertions given the ontology, in particular its classes and named individuals, are generated because the TREETRAVERSE procedure is in its essence a power set generating procedure. On each step of TREETRAVERSE, a class assertion $C(u)$ is generated, and then the procedure continues to generate a combination with $C(u)$ (if the ontology remains consistent) and without $C(u)$. Furthermore, in combination with the NEXTINDIVIDUAL procedure, it is ensured that all of the named individuals and classes in the ontology are included.

Because all of the possible combinations are generated, and any combination not a subset of another combination is stored as shown in Lemma 3, the theorem holds.

5.2.6 Complexity Analysis

This section analyses the complexity of the algorithm developed. For more information on complexity theory, see a book that covers the topic such as [39]. Due to the combinatorial nature of the world concretization problem, the worst case complexity of any algorithm solving it is rather dire. The worst case complexity of computing a power set is itself $O(2^n)$ (EXPTIME), as $|\mathcal{P}(C)|$ is $2^{|C|}$. The naive algorithm computes the power box giving $n = |C| * m$, where C is the set of classes in the class hierarchy and m is the number of individuals. The optimized algorithm has the same complexity of its traversal, as in the worst case all of the combinations that are possible to generate will be legal.

A reasoner must be used to remove inconsistent ontologies from the power box after the naive algorithm is ran, and to discard inconsistent ontologies on-the-fly in the optimized algorithm. As such, both of the algorithm's complexities depends on the chosen OWL profile's complexity for checking ontology consistency. In our case, the direct semantics of OWL 2 is used and the inputs are the axioms and assertions of the ontology. With these specifications, the ontology consistency reasoning is N2EXPTIME-complete [27]. Thus both the naive and optimized algorithm are N2EXPTIME-complete when the OWL 2 Direct Semantics is used. The complexity of the less expressive languages is either PTIME-complete or in AC^0 (which is more efficient than LOGSPACE). Figure 5.20 shows the languages, their complexities for checking consistency and how the algorithms' complexities are affected. The first row states the complexity of the algorithms' traversal without calls to a reasoner.

5.3 Summary

We have presented a methodology for modeling worlds in DL, and an algorithm which uses the OWL API to find the consistent maximal class assertion combinations of an OWL ontology. In the next chapter we apply the modeling methodology on

OWL Language	Consistency Complexity	Algorithm Complexity
—	—	EXPTIME
OWL 2 Direct Semantics	N2EXPTIME-complete	N2EXPTIME-complete
OWL 2 EL	PTIME-complete	EXPTIME
OWL 2 QL	In AC^0 (LOGSPACE)	EXPTIME
OWL 2 RL	PTIME-complete	EXPTIME

Figure 5.20: Complexities of concretizing algorithm given OWL language

the submarine fan use case. Then, in Chapter 7, we discuss the results of running the algorithm developed on the use case ontology.

Chapter 6

Underdetermined Worlds in the Geological Assistant

In this chapter we show how the modeling methodology and algorithm from the previous chapter can be applied in the Geological Assistant pipeline presented in Section 1.2 on page 4. In the Geological Assistant, static geological data is used in generation of scenarios. Some of the static data are underdetermined, and thus must be concretized to enable the geological scenario reasoning. This could be done as a part of the dynamic reasoning process, but we show an approach where the processing of the underdetermined static data is done before the scenario generation is started. This approach might be preferable due to modularization of the system, and utilizing the strengths of different tools.

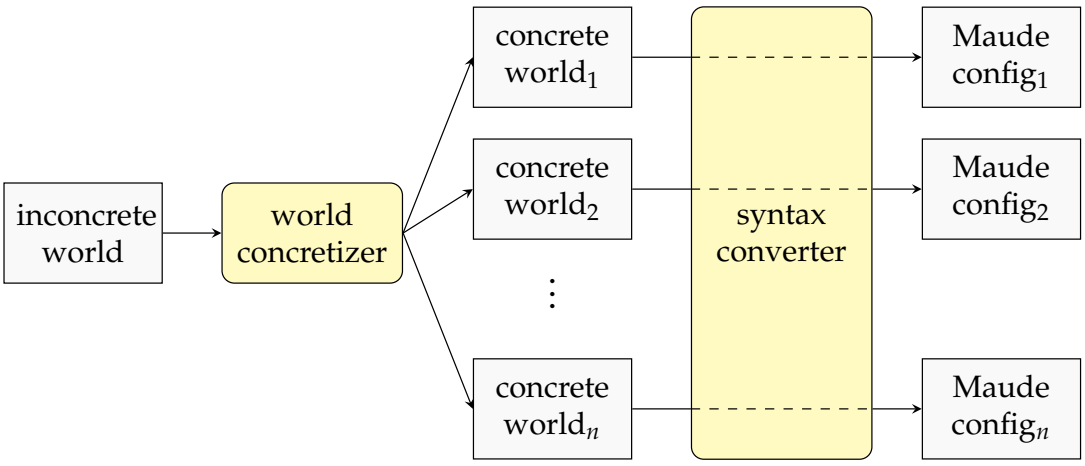


Figure 6.1: Scenario pre-processing of an underdetermined world

The complete scenario pre-processing module is visualized in Figure 6.1. Note

that the only technical specificality in the figure is the Maude configurations. The rest of the system can be implemented however one would like, but the goal of the application is to produce Maude configurations. The world concretizer is implemented with an implementation of the consistent maximal ABox algorithm presented in the previous chapter, while the syntax converter is presented in Section 6.3. The task of the syntax converter is to convert the ABoxes output by the concretizer to Maude configurations. The gray boxes in the figure are files representing worlds, and the yellow boxes are procedures operating on these files. An inconcrete world has several possible instantiations which are generated by the concretizer. The concrete worlds and Maude configurations are different representations of the instantiated worlds, visualized by the dashed lines through the syntax converter.

When creating the syntax converter, there are two important factors: (i) how the ABoxes are serialized, and (ii) a mapping of OWL entities to Maude entities. The first factor is important for writing the inputs of the translation rules generating Maude configurations. The second factor is important for deciding how the inputs are represented in Maude. We look at both of these factors in Section 6.2, but first we look at how the use case is modeled in OWL. Lastly, we present the implementation of the syntax converter.

6.1 Use Case Ontology

The use case ontology is a representation of a world. Its TBox describes the objects and how they can relate to each other, while the ABox represents a specific underdetermined world. The ABox is sent into the concretizing algorithm, and concrete worlds are generated in OWL syntax.

To model the use case, we model the formalizations from Section 3.4 in OWL. It is quite simple for faults and sealing capacities, as was shown in Section 5.1.2. However, for the logical distribution of components in the submarine fan it is more complex due to the order dependencies. We have to include a relation between rock units showing their location ordering from first to last. Further, we must formalize the legal environment orders.

With the object modeling methodology shown in the previous chapter, we got almost all of the building blocks we need to formalize the use case in OWL. What we lack is how to represent the ordering of the environments. To do this, we add a property $frontOf$, signifying that an individual is in front of another individual. Recall the lobe ordering formalization, i.e.

$$Lobe \rightarrow \{Lobe, LobeFringe\}.$$

GeoUnit Object Axioms
$GeoUnit \equiv Location \sqcap Porosity \sqcap Permeability$
$GeoUnit \sqsubseteq \neg SealingCapacity$
$Fault \equiv SealingCapacity$
$Fault \sqsubseteq \neg Location \sqcup \neg Porosity \sqcup \neg Permeability$

Value Set Axioms
$SealingCapacity \equiv Sealing \sqcup NonSealing$
$Porosity \equiv Porous \sqcup NonPorous$
$Permeability \equiv Permeable \sqcup NonPermeable$
$Location \equiv FC \sqcup IC_1 \sqcup DC \sqcup IC_2 \sqcup L \sqcup LF \sqcup BP$

Dependency Axioms
$FC \sqsubseteq \forall frontOf.FC \sqcup \forall frontOf.IC_1 \sqcup \forall frontOf.DC$
$IC_1 \sqsubseteq \forall frontOf.IC_1 \sqcup \forall frontOf.DC$
$DC \sqsubseteq \forall frontOf.DC \sqcup \forall frontOf.IC_2 \sqcup \forall frontOf.L$
$IC_2 \sqsubseteq \forall frontOf.IC_2 \sqcup \forall frontOf.L$
$L \sqsubseteq \forall frontOf.L \sqcup \forall frontOf.LF$
$LF \sqsubseteq \forall frontOf.LF \sqcup \forall frontOf.BP$
$BP \sqsubseteq \forall frontOf.BP$

Object Property Axioms
$\top \sqsubseteq \leq 1\ frontOf\ Thing$ (functional)
$\top \sqsubseteq \leq 1\ frontOf^{-}\ Thing$ (inverse functional)
$\top \sqsubseteq \neg \exists frontOf.Self$ (irreflexive)

Figure 6.2: Abbreviated use case TBox $\mathcal{T}_U$

This dependency can be modeled in DL using the new property and inclusion as follows

$$Lobe \sqsubseteq \forall frontOf.Lobe \sqcup \forall frontOf.LobeFringe$$

This will make sure that if an individual is asserted to be a *Lobe* and $frontOf$-related to another individual, what it is in front of must be either of type *Lobe* or of type *LobeFringe*. The object class *GeoUnit*, representing geological units, is set as the disjoint union of all the locations.

Figure 6.2 shows the most interesting parts of the use case TBox, $\mathcal{T}_U$.[1] The parts left out of the TBox are the disjointness axioms of the value set classes, e.g. $Sealing \sqcap NonSealing \sqsubseteq \bot$, and the specification of permeability and porosity of the

[1] See Figure A.1 on page 96 for the rest of TBox $\mathcal{T}_U$

$Disjoint(frontOf, frontOf^-)$ (asymmetric)

Figure 6.3: Use case RBox $\mathcal{R}_U$

locations. In $\mathcal{T}_U$ we see that *SealingCapacity* is a value set with no dependencies, as it has atomic classes on the right-hand side of its equivalency. However, *Location* is a value set with dependencies, as the classes on the right-hand side of its equivalency are value restrictions. The object property axioms in the end state that the $frontOf$ property is functional, inverse functional and irreflexive. The RBox $\mathcal{R}_U$ of the use case is shown in Figure 6.3. It only contains an axiom stating that $frontOf$ is asymmetric. Figure 6.4 shows the use case's ABox $\mathcal{A}_U$, which in combination with $\mathcal{T}_U$ and $\mathcal{R}_U$ is a representation of the *UseCase* world.

$\{Fault(f0)$,	$Fault(f1)$,	$Fault(f2)$,
$Fault(f3)$,	$GeoUnit(gu5)$,	$GeoUnit(gu8)$,
$GeoUnit(gu11)$,	$GeoUnit(gu14)$,	$frontOf(gu5, gu8)$,
$frontOf(gu8, gu11)$,	$frontOf(gu11, gu14)\}$	

Figure 6.4: Use case ABox $\mathcal{A}_U$

6.2 Mapping OWL to Maude

To formalize the conversion of worlds in OWL syntax to Maude syntax, we define a mapping from OWL to Maude through the abstraction of worlds as presented in Chapter 4. The mapping of OWL to Maude is done by finding appropriate entities in Maude to represent the entities of a world. Then each of the Maude entities are mapped to the corresponding OWL entity.

Naturally, we use the Maude sort Oid to represent an object's *id* and binary operators to represent binary relations. Then attribute values of an object in Maude represent attribute values in a world. Maude object names are used to represent the object names, and the name of a Maude object attribute represents the name of a value set. The complete mapping is shown in Figure 6.5. Any entity not in the figure is not a part of the mapping, as it is not necessary for the representation of a world.

In Figure 6.6 on the facing page we show a concrete mapping example of two geological units gu_1 and gu_2 with the attribute *Location*, and where the first is located in front of the other. Further, the first is located in a *Lobe* and the second is located at a *LobeFringe*. The left-hand side of the figure is the OWL ABox, and the right-hand side of the figure is the resulting Maude configuration.

OWL		World		Maude
Individual Name	→	Object ID (*id*)	→	Oid
Object/Data Properties	→	Binary Relations	→	Binary Operators
Value Class	→	Value Name	→	Object Attribute Value
Value Set Class	→	Value Set Name	→	Attribute Name
Object Class	→	Object Name	→	Object Name

Figure 6.5: OWL entities mapped to Maude entities

OWL ABox		Maude Configuration
$GeoUnit(gu_1)$, $Location(gu_1)$, $Lobe(gu_1)$	→	< gu_1: GeoUnit | Location: lobe >
$GeoUnit(gu_2)$, $Location(gu_2)$, $LobeFringe(gu_2)$	→	< gu_2: GeoUnit | Location: lobeFringe >
$frontOf(gu_1, gu_2)$	→	frontOf(gu_1, gu_2)

Figure 6.6: Example inputs and outputs of conversion

As Maude is more expressive than OWL, mapping from OWL to Maude is easier than vice-versa. If automatic conversion of Maude configurations to OWL ontologies is added, Maude and OWL could communicate during scenario generations. This could be interesting in case the scenarios generated involve different static facts where an OWL ontology and reasoner could be used to infer new knowledge. Even though it is an interesting, we do not cover the topic of Maude configuration to OWL ontology conversion in this thesis. However, it should be possible to expand upon our work to find a way to convert fragments of Maude configurations to OWL ontologies.

6.3 Syntax Conversion

Now we show code listings of a syntax converter implemented in Maude.[2] We show the conversion of a concretized *GeoUnit* object with the three attributes *Porous*, *Permeable* and *FeederChannel* from OWL to Maude. The code shown serves as an example for conversions of other concretizations. We choose to implement the converter in Maude mainly because an ABox is similar to a Maude configuration, and this makes it simple to define the elements of an ABox in Maude. Also, Maude allows for an elegant and readable implementation.

From the mapping in Figure 6.5, the object id is represented with the predefined

[2]See Listing B.1 on page 99 for the complete module

sort `Oid`, and binary relations are represented by binary operators of sort `Msg`. The three other entities need their own sorts. For values we define the sort `Value`. For object names we define `ObjectName`. Finally, for value set name we define `ValueSet`. All of these three are subsorts of `OWLClass`, which again is subsort of `Msg`. These declarations are used to enable the conversions, and they are shown in Listing 6.1. The sort `OWLClass` is used for all of the class assertions.

Listing 6.1: Sorts for OWL entities

```
sorts OWLClass Value ValueSet ObjectName .
subsorts Value ValueSet ObjectName < OWLClass .
subsort OWLClass < Msg .
```

The next point is to declare constants which represent the different types of class assertions. This is done in Listing 6.2. These are declared in one of two ways. The object names and value set names are simply defined as constants of sort `ObjectName` or `ValueSet`. For the values, we need to enable their conversion to Maude constants. To do this, we define new sorts representing variables connected to value sets, and declare these as subsorts of `Value`. For example, the sort `OWLLocation` is used for any element in the value set of *Location*. Later, we declare the sort `ObjectAttribute` which represents Maude constants used as attribute values, and `Value` is connected to it through a function `convert`.

Listing 6.2: Declaration of OWL classes

```
sorts OWLLocation OWLPermeability OWLPorosity .
subsort OWLLocation OWLPermeability OWLPorosity < Value .

--- values
op FeederChannel : -> OWLLocation .
op Permeable : -> OWLPermeability .
op Porous : -> OWLPorosity .

--- object names and value sets
op GeoUnit : -> ObjectName .
op Location : -> ValueSet .
op Permeability : -> ValueSet .
op Porosity : -> ValueSet .
```

It is important to note a design decision regarding the serialization of an ABox. Instead of serializing class assertions directly as shown in Figure 6.6, for example $Permeable(gu_1)$, we serialize them with a binary relation `type`. The first element in `type` is the class name, and the second element is the individual name of the assertion. For example `type(Permeable, "gu1")`. This is done specifically for

utilizing variables in Maude when converting from `Value` to `ObjectAttribute` as we show later in Listing 6.6.

Now we come to the target entities of the conversion. Firstly, we define some constants used as values for object attributes. We define sorts `Location`, `Permeability` and `Porosity` which are all subsorts of `ObjectAttribute`.

Listing 6.3: Maude constants (targets for `Value` constants)

```
sorts ObjectAttribute LocationT PermeabilityT PorosityT .
subsorts LocationT PermeabilityT PorosityT < ObjectAttribute .

op feederChannel : -> LocationT .
op permeable : -> PermeabilityT .
op porous : -> PorosityT .
```

Next, we declare a function `type` as a `Msg` with a `Value` argument and an `Oid` argument.

Listing 6.4: Type operator

```
--- operator to construct maude objects from an OWL message
op type : OWLClass Oid -> Msg .
```

To make use of `type`, we need an object or a binary relation we can convert something into. We define the object `GeoUnit` as follows.

Listing 6.5: Geological Unit object declaration

```
op <_: GeoUnit | Permeability:_, Porosity:_, Location:_> :
    Oid PermeabilityT PorosityT LocationT -> Object [ctor] .
```

With the `GeoUnit` object and `type` operator, we define an equation that converts a set of OWL classes into a Maude object as shown in Listing 6.6. Here we can see the utility of the `type` operator and the subsorts of `Value`, as they let us use variables to generalize the conversion.

Listing 6.6: Equation converting object from OWL to Maude object

```
var O : Oid .
var PB : OWLPermeability .
var PR : OWLPorosity .
var L : OWLLocation .

eq type(GeoUnit, O) type(Permeability, O) type(PB, O) type(Porosity, O)
type(PR, O) type(Location, O) type(L, O)
  =
```

```
   < O : GeoUnit | Permeability: convert(PB), Porosity: convert(PR),
      Location: convert(L) > .
```

For the conversion to work, we need to define the `convert` equation. The purpose of `convert` is to translate an OWL (value) class into an attribute in Maude. Listing 6.10 shows the specific conversion from `FeederChannel` to `feederChannel`. It is the bridge that connects the whole conversion process. If this type of equations including all of the other declarations shown above are added for all of the OWL syntax, then a serialized OWL ABox is convertible to Maude syntax and can be used in Maude computations.

Listing 6.7: Conversion of values to constants

```
op convert : Value -> ObjectAttribute .
eq convert(FeederChannel) = feederChannel .
eq convert(Permeable) = permeable .
eq convert(Porous) = porous .
```

A serialization of the ABox includes the class assertions in the `type` operator, and looks as follows.

Listing 6.8: Serialized ABox

```
mod OWL-ABOX is
  protecting OWL-CONVERTER .
  op world : -> Configuration .
  eq world = type(GeoUnit, "gu5") type(Permeability, "gu5")
  type(Permeable, "gu5") type(Porosity, "gu5")
  type(Porous, "gu5") type(Location, "gu5")
  type(FeederChannel, "gu5")
endm
```

By running the syntax converter, we get the following result.

Listing 6.9: End result of conversion

```
rewrite in OWL-ABOX : unknowns .
rewrites: 6 in 0ms cpu (0ms real) (~ rewrites/second)
result Object: < "gu5" : GeoUnit | Permeability: permeable,
Porosity: porous, Location: feederChannel >
```

To enable conversion of more concretized *GeoUnit* objects, we simply add declarations of more value classes and add them to the `convert` equation. We must also add their target constants.

Listing 6.10: Conversion of additional values to constants

```
--- OWL classes (inputs)
op InterChannel1 : -> OWLLocation .
op InterChannel2 : -> OWLLocation .
op DistributaryChannel : -> OWLLocation .
op NonPermeable : -> OWLPermeability .
op NonPorous : -> OWLPorosity .

--- Maude constants (outputs)
op interChannel : -> LocationT .
op distributaryChannel : -> LocationT .
op non-permeable : -> PermeabilityT .
op non-porous : -> PorosityT .

--- Conversions
eq convert(InterChannel1) = interChannel .
eq convert(InterChannel2) = interChannel .
eq convert(DistributaryChannel) = distributaryChannel .
eq convert(NonPermeable) = non-permeable .
eq convert(NonPorous) = non-porous .
```

This ends the presentation of the syntax converter. It works as an example that can be followed to convert any world modeled in OWL to a Maude configuration for further generation of scenarios.

6.4 Summary

We have shown a framework where a world concretizer and a syntax converter is used together to concretize an underdetermined world modeled in OWL and convert the resulting concrete worlds to Maude configurations. This finishes the presentation of the world concretization problem and how we solved it. In the next chapter we evaluate our approach.

Part III

Discussion and Conclusions

Chapter 7

Discussion

In this chapter, we evaluate the approach presented in the previous part of the thesis. In particular, we evaluate the OWL world modeling and concretization methodology from Chapter 5. In addition, we propose some ways the approach can be improved and a different application area for it.

To evaluate the approach, we view it from the perspective of the potential users and developers of a system implementing it. We pose the following 4 questions to guide the topics of the evaluation.

1. How is it to use for an end-user?

2. How is it to maintain and modify for a developer?

3. How efficient is it?

4. Does it work? Are the results generated correct?

To answer these questions, we discuss modularity which we argue in our case is relevant to the first two questions. Then, the usability of the modeling is briefly reviewed. For efficiency we discuss complexity and give some of our experience during testing. To discuss correctness, we argue that the algorithm developed makes sure the results are correct as long as the ontology is modeled correctly. We then discuss two topics unrelated to the questions above. The first is a discussion of ways the computation could be optimized. The second is a brief look at using the consistent maximal ABox algorithm for debugging the semantics of an ontology.

Before we enter into the topics above, we briefly present an alternative approach which was developed in an earlier version of the world concretizer. We do this to highlight some of the benefits of the approach we developed for this thesis.

7.1 Comparison

We compare the approach presented in the previous chapters with an imperative programming approach where the algorithmic control is intertwined with the formalized domain knowledge. First, the intertwined approach is presented.

7.1.1 Intertwined Domain Knowledge and Algorithmic Control

Now we present two algorithms concretizing worlds following an approach where domain knowledge and algorithmic control are intertwined.[1] In other words, where the domain knowledge is hard coded into the source code. The algorithms are abstracted from a Python script, which was an early prototype of the world concretization problem in the Geological Assistant system.

The algorithms computes the value assignment trees of each type of variable (attribute) separately, then combines the value assignments for the different variables in the end to generate a concrete world. The first algorithm's parameters are a number *count*, the number of objects that have the attribute, and a list *valueset*, the value set of the attribute. The value assignments are represented as a list where the first element in the list represents the value assignment for the attribute of the first object, the second for the second object, etc.

The value assignment of objects' variables without any dependencies is computed with the procedure CONCRETIZER in Algorithm 3. It is a rather straightforward algorithm which generates all permutations of the input *valueset* for *count* number of times. The algorithm is run independently for each attribute without dependencies, and then the results are combined, for example with the procedure COMBINE in Algorithm 4.

At first glance, it does not look like there is any domain knowledge represented in Algorithm 3. It takes some inputs and recursively generates all of the combinations. However, the procedures represent that the values generated are independent. This algorithm is not able to generate value assignments for attributes with dependencies. So when there is a dependency in the world, an algorithm specifically for that dependency must be added.

Assignment with Object Dependencies

When adding support for value assignment of locations, one has to include both how to compute the value assignments as well as how to represent the domain knowledge within the program. The procedures in Algorithm 5 on page 80 shows one way of computing value assignments for the location attribute. The algorithm mimics the

[1] Implementation at `https://github.com/vskaret/PythonScenarioExpansion`

Algorithm 3 Independent Variable Value Assignment

```
1: procedure CONCRETIZER(count, valueset)
2:     global concretizations = new list()
3:     CONCRETIZE(count, valueset)
4:     return concretizations
5:
6: procedure CONCRETIZE(count, valueset, concretization)
7:     if |count| == 0 then
8:         concretizations.append(concretization)
9:         return
10:    for all value ∈ valueset do
11:        copy = concretization.copy()
12:        copy.append(value)
13:        CONCRETIZE(count − 1, valueset, copy)
```

Algorithm 4 Value Assignment Combiner

```
1: procedure COMBINE(concretizations1, concretizations2)
2:     combined = new list()
3:     for all c1 ∈ concretizations1 do
4:         for all c2 ∈ concretization2 do
5:             concretization = c1.append(c2)
6:             concretization.append(concretization)
```

formalization of the environment orders in Figure 3.6 on page 34 by having a list *ordertuples*. An order tuple is a pair where the first element is a location l, and the second element is a list of locations that can come after l. The *ordertuples* work as a combination of a value set and the formalization of how the order of the locations should be. Listing 7.1 shows a representation of such a list in Python where each variable is a string (e.g. FC = "feederChannel").

Listing 7.1: List of tuples representing the location order in Python

```
ordertuples = \
[(FC, [FC, IC_1, DC]), # FC -> {FC, IC_1, DC}
(IC_1, [IC_1, DC]),
(DC, [DC, IC_2, L]),
(IC_2, [IC_2, L]),
(L, [L, LF]),
(LF, [LF, BP]),
(BP, [BP])]
```

Algorithm 5 Dependent Location Value Assignment

```
1:  procedure CONCRETIZER(count, ordertuples)
2:      global concretizations = new list()
3:      while |ordertuples| > 0 do
4:          term = ordertuples[0][0]
5:          concretization = new list()
6:          concretization.append(term)
7:          CONCRETIZE(count, ordertuples, term, concretization)
8:      return concretizations
9:
10: procedure CONCRETIZE(count, ordertuples, current, concretization)
11:     remaining = tuples.copy()
12:     first = ordertuples[0][0]
13:     while first ≠ current do
14:         ordertuples.pop(0)
15:         first = ordertuples[0][0]
16:
17:     if count ≤ 1 then
18:         concretizations.append(concretization)
19:         return
20:
21:     neworder = remaining.copy()
22:     for all term ∈ remaining[0][1] do
23:         if term ≠ current then
24:             neworder.pop(0)
25:
26:             newconc = concretization.copy()
27:             newconc.append(term)
28:             CONCRETIZE(count − 1, neworder, term, concretization)
```

Now we turn to a comparison of the approach presented here, where the domain knowledge is hard coded into a program, with the approach using an ontology shown earlier in the thesis.

7.1.2 Modularity

Modularization [32] is a way of designing software systems such that the system is divided into distinct modules, where each module is assigned a responsibility. Among the expected benefits are more flexibility of code development and better

comprehensibility of the system. We argue that our modular framework of generating concrete worlds is an example showing these benefits.

In the Scenario pre-processing framework presented in Chapter 6 there are two modules. The world concretizer and the syntax converter. As their names suggest, they are responsible for different tasks. The concretizer is further modularized by using a module for verifying consistency of ontologies (legality of value assignment). We now compare the effects of the modularization in terms of flexibility and comprehensibility.

Flexibility

First of all, using OWL and the open source OWL API gives us flexibility in the sense that our development only has to focus on how to utilize features of OWL to compute the results. There is no code development involved with formalizing the knowledge of the domain. As a consequence, programmers working with the implementation of the algorithm do not need to worry about formalizing the knowledge correctly. Furthermore, domain experts formalizing knowledge do not need to see nor write any code. With the intertwined approach, programmers and domain experts would both have to be involved with writing code and formalizing knowledge.

By having a modular implementation, developers are able to work on separate parts of the implementation at the same time. For example, some can work on improving the concretization algorithm while others work on the syntax converter. In an intertwined approach, where value assignments and syntax is generated at the same time, this would not be possible. The syntax would be a part of the value assignment procedure, and changing the syntax might lead to problems with the value assignment computation. By using the OWL API with a module with wide open source support such as the reasoner, the flexibility goes even further as there are different implementations of the reasoner available instantly. Furthermore, an implementation could be changed without our knowledge of what is changed, as long as it still carries out its reasoning responsibility.

Comprehensibility

Even for a competent programmer, who only has a little knowledge of Description Logic, it is most likely easier to read and understand the knowledge represented in the Use Case ontology in Chapter 6 than the algorithms presented previously in this chapter. This is due to Description Logic being a language which is purely focused on knowledge representation, it does not describe the computation at all. In fact, for our purpose it is somewhat similar to declarative programming. This transparency in the formalized domain knowledge is an important factor for systems

where people who might have no programming experience are involved in verifying that the domain is modeled correctly.

Further, by having only one general algorithm for computation, there is less room for bugs in the implementation. Actually, we discovered a bug in our Python scripts through the results generated by the OWL method. A few concrete worlds were removed due to an attempt at optimizing the code by removing the duplicate cases of locations with only InterChannels. Although there is less room for bugs in the implementation of the algorithm computing the value assignments, a new sort of bug where the ontology itself is wrong is introduced. Even so, it is likely easier to pinpoint the source of the problem when one only has to work with a knowledge representation language.

7.1.3 Ease of Use

We now look at how the two different approaches compare when it comes to formalizing new knowledge. When the domain knowledge and algorithmic control are separated, the system is not only flexible in the sense that different developers can work on different parts on the system. By using OWL as the modeling language, users with no programming experience can model worlds with the system. This flexibility is beneficial when the target users of the system are domain experts who might have little to no programming experience. Moreover, by using OWL for the knowledge representation in the framework, one gets the added benefit of already implemented GUIs such as Protégé[2] [29] to do the modeling in. In other words, instead of using the programming language as an interface to model the domain knowledge, OWL with its available interfaces are used.

The more complex the dependencies of a world gets, the more difficult it becomes to formalize the knowledge with both approaches. However, with the intertwined approach one must also find the correct approach to compute it. This is likely to also get more difficult as the formalization complexity increases. With the formalization kept by itself when using OWL, this is not an issue. Furthermore, the ontology reasoner lets one know if the formalization is inconsistent. This will prohibit some bugs where the formalization itself is incorrect.

For example, adding a new object to the Use Case ontology such as a trap does not seem too daunting. A reservoir trap is a geological unit where the accumulation of hydrocarbons is possible. In our use case, we could model a trap as a geological unit which is porous and permeable and which is in front of a sealing fault. Something like the following axiom could work.

$$Trap \equiv GeoUnit \sqcap NoMigration \sqcap \neg NonPorous \sqcap \neg NonPermeable,$$

[2] `https://protege.stanford.edu/`

where *NoMigration* represents that a migration is not possible with the following (disjoint) union.

$$NoMigration \equiv \forall fOf.Sealing \sqcup \forall fOf.NonPorous \sqcup \forall fOf.NonPermeable,$$

where fOf is short-hand for the $frontOf$ property.

Note that for brevity the *Trap* axiom does not strictly follow the modeling approach, and the disjoint axioms for the attributes of the *NoMigration* value set are left out. Further, axioms formalizing that faults have a location just like geological units would have to be added, and in the ABox one would have to include the faults in the $frontOf$ property assertions. Also note that these axioms are not tested, but it is conceivable that they, or something similar, would work. To formalize the same knowledge with the intertwined approach seems like a more daunting task, because now there are objects which can be both traps and geological units at the same time.

7.1.4 Complexity

The approach for concretizing worlds presented in this thesis has an extreme complexity. It requires the use of axioms of the type disjoint classes and disjoint union of classes for the modeling, and this is only supported by the direct OWL 2 semantics where the complexity of ontology consistency reasoning is N2EXPTIME-complete. This has to be far worse than procedures that both formalize and generate the value assignments, where the complexity most likely is not worse than EXPTIME.

An important issue we were unable to solve in this thesis work is whether it is possible to model a world in OWL with any of the lesser expressive OWL languages. With any of these, the complexity will go down to EXPTIME, which is the lowest one can get with combinatorial problems like the world concretizations. The reason for this being the lowest worst case complexity is due to the fact that all of the combinations must in the worst case be investigated.

7.2 Correctness of Results

For the method developed to be correct, all possible concrete worlds given the dependencies must be generated from an ontology. This means that the world modeling methodology and the maximally consistent ABox algorithm combined must solve the issue. In Section 5.2.5 on page 59 we argued for why the results of the algorithm are sound and complete. For the correct results to be generated then, the ontology must correctly describe a world such that it is a correspondence

1. $\{NonSealing(f3), NonSealing(f1), NonSealing(f2)\}$
2. $\{NonSealing(f3), Sealing(f1), NonSealing(f2)\}$
3. $\{Sealing(f3), NonSealing(f1), NonSealing(f2)\}$
4. $\{Sealing(f3), Sealing(f1), NonSealing(f2)\}$
5. $\{NonSealing(f3), NonSealing(f1), Sealing(f2)\}$
6. $\{NonSealing(f3), Sealing(f1), Sealing(f2)\}$
7. $\{Sealing(f3), NonSealing(f1), Sealing(f2)\}$
8. $\{Sealing(f3), Sealing(f1), Sealing(f2)\}$

Figure 7.1: Class assertion combinations generated for world *Faults*

between the consistent maximal ABoxes and the concrete worlds. For independent value sets this is simple, but for attributes with dependencies it is more difficult as the dependencies must be modeled correctly.

7.2.1 Example Results

Before evaluating the results of the whole use case, we present results from the examples visualized with the value assignment trees in Chapter 4. First, we had the world *Faults* with faults f_1, f_2 and f_3 with the attribute $sealingCapacity \Leftarrow \{Sealing, NonSealing\}$. The concrete worlds generated by the world concretizer for the underdetermined world *Faults* is shown in Figure 7.1 where only class assertions representing values are kept for brevity. The order of the combinations are in the order they were generated. The reason why the correct results are generated is because of the disjoint union as mentioned in the end of Section 5.1.1 on page 46.

Then we had the more interesting world $Rocks_R$ with geological units gu_5, gu_8 and gu_{11} with the attribute $location \Leftarrow \{Lobe, LobeFringe, BasinPlain\}$ and the relation $frontOf$. In the value assignment tree in Figure 4.2 on page 41, we saw that from this underdetermined world, 8 concretized worlds could be generated. When the world concretizer is run, the 8 combinations shown in Figure 7.2 are generated.

The reason why these results are generated correctly is first of all because *Location* is set to be the disjoint union of a set of locations. Each of the locations are then set to be the union of value restrictions with the $frontOf$ property, which represents that a unit is directly in front of another unit. When it is asserted in the ABox that an instance of a *GeoUnit* is $frontOf$ related to another individual, the other individual must then be of one of the classes in the value restriction. Otherwise the ontology would be inconsistent.

1. $\{Lobe(gu5), Lobe(gu8), Lobe(gu11)\}$
2. $\{Lobe(gu5), Lobe(gu8), LobeFringe(gu11)\}$
3. $\{Lobe(gu5), LobeFringe(gu8), BasinPlain(gu11)\}$
4. $\{Lobe(gu5), LobeFringe(gu8), LobeFringe(gu11)\}$
5. $\{BasinPlain(gu5), BasinPlain(gu8), BasinPlain(gu11)\}$
6. $\{LobeFringe(gu5), BasinPlain(gu8), BasinPlain(gu11)\}$
7. $\{LobeFringe(gu5), LobeFringe(gu8), BasinPlain(gu11)\}$
8. $\{LobeFringe(gu5), LobeFringe(gu8), LobeFringe(gu11)\}$

Figure 7.2: Class assertion combinations generated for world $Rocks_R$

7.2.2 Use Case Results

The computation of the *UseCase* ontology turned out to be too intensive for the hardware used to be able to finish it in reasonable time for this thesis. After a few days only around 300 of several thousand concrete worlds were generated. Because of this, we modified the ontology such that *GeoUnit* only had the *Location* attribute.[3] Further, during the testing, one of the faults were left out of the ontology. The actual input was Figure 6.4 on page 68 without $Fault(f0)$. This means that the amount of concrete worlds halved as the faults only has one underdetermined attribute with a binary value set. The amount of concrete worlds generated was then 536, and now we briefly discuss why this is correct.

We have seen that the 3 faults by themselves generate 8 concrete worlds. Generating concrete worlds with 4 geological units with only the *Location* attribute generates 67 concrete worlds. It is difficult to calculate how many it should be due to the dependencies, but going through them all manually shows it is correct.[4] The point is that the modified use case should generate 8 (concrete faults) * 67 (concrete geological units) = 536 (concrete worlds). If interested, the generated results can be verified by running the implementation.[5]

Even though we reduced the size of the input for the computation, the concretization of the modified *UseCase* still took more than 72 hours on a personal computer with a decent CPU (i5-2500K @ 3.30GHz x 4) and 12GB allocated to the JVM heap. When executing a computation is this time-demanding, there is likely room for optimizations.

[3] See Figure A.2 on page 97 for the TBox used in the testing

[4] See Figure A.3 on page 98 for a list of location orders with 4 geological units

[5] `https://github.com/vskaret/consistent-maximal-abox-generator`

7.3 Improving Efficiency of Computations

With the poor complexity of the algorithm, its search space quickly explodes when the size of input is increased. This is true even if the complexity is improved to EXPTIME. Thus optimizations that makes the search space for a process smaller will likely have a big effect on the time spent computing. We look at two approaches of achieving this: (i) by changing the algorithm and (ii) by changing the ontology modeling.

7.3.1 Algorithm

We suggest two ways of changing the algorithm to lower the computing time. The first is to add a way to specify which individuals to include in the concretization. The second is to parallelize the algorithm.

Partial World Concretizations

In the algorithm, every attribute of every object is concretized to generate the concrete worlds. However, one can imagine cases where there are only the concretizations of a specific part of the world which is of interest. For example, when figuring out which new graphic card and hard drive to purchase for your computer, you might be most interested in which graphic cards are supported by the computer. In this case, the computation can be sped up by limiting the search space to only concretizing graphic cards. To achieve this, a reserved class name in OWL can be used. For example, a class *Unknown* can be used to specify for the algorithm which individuals to concretize.[6] When there are many objects in the world, but most of them are not interesting, the speed of the process should improve significantly. There might also be situations with objects which are already concretized, then it is unnecessary to run the individual through the algorithm.

Parallelization

We briefly discuss an approach to parallelize the algorithm that could speed up the computation time. For a description of parallel programming techniques, see for example [35].

The algorithm mainly performs two tasks during computation: (i) it checks the consistency of an ontology and (ii) it checks whether or not the combination found is maximal. If the algorithm was to be parallelized, only the second task requires any communication between the threads. The first does not require communication

[6]This feature is included in our implementation with the reserved class name Unknown

as all of the data is available in the thread's ontology object. Thus each thread can find its local consistent maximal combinations, and then the local combinations can be compared in the end.

If the programmer keeps track of which thread is handling which parts of the search space, it is possible to minimize the time used for the local comparison of combinations in the end as follows. Assume there are two threads t_1 and t_2. Thread t_1 is tasked with the search space which consists of the largest ABox combinations. In other words, it will include the first class assertion in its local combinations, and t_2 will not include the first class assertion. Then, at the end of the parallel computation, one can assume all of the combinations in t_1 are maximal, and if any of t_2's combinations are maximal depends on them not being subsets of t_1's combinations. This can be generalized with more threads, it is just that the combinations in the end must be checked in the right order (from low to high assuming t_1 gets the search space with the largest combinations and t_n gets the search space with the smallest combinations).

As the search space quickly becomes large due to the computational complexity, the cost of starting new threads is likely very low, so making the algorithm concurrent will in most cases be very beneficial for the speed of the computation. As the threads can compute their full search space concurrently, and the critical region is not until the end when comparing the results of each thread, the parallelization will also not suffer poor efficiency due to synchronization.

7.3.2 Modeling Methodology

The number of axioms in the ontology is the size of the input to the algorithm. As the worst case complexity is poor, it is beneficial to keep the size of the input small. Here we suggest two approaches to achieve this: (i) where each independent part of the ontology is separated into its own ontology and (ii) by adding trigger rules.

Local Concretizations

If there are objects in a world that are not dependent on each other, their value assignments can be computed separately. For example, the value assignment of a fault in our use case did not depend on any other objects. By splitting the use case ontology into several ontologies, it should generate each local value assignment at a fragment of the time it takes for the whole ontology. This is true no matter the expressiveness of the OWL language used as the traversal of the algorithm is either way EXPTIME. The local ABoxes could be combined with ABoxes from the other ontologies to generate all of the concrete worlds of the use case. Following this approach allows computations to be ran on separate hardware, which could improve the execution time further.

Trigger Rules

Trigger rules of the form $C \Rightarrow D$ works similarly to implication in that if and individual is shown to be an instance of C, it is also an instance of D [3]. These can be used to split the class hierarchy of the ontology into classes that the algorithm generate combinations of, and classes that are a consequence of the classes in the combinations. For example, if we assume a geological unit which is a sandstone and located in a lobe is porous, then we can add the trigger rule $Sandstone \sqcup Lobe \Rightarrow Porous$. If we model $Porous$ and $NonPorous$ like this, they can be removed from the class hierarchy the algorithm traverses and be put into a separate class hierarchy. By reducing the amount of nodes in the tree-traversal like this, the overall time taken for a computation will also be reduced.

7.4 Ontology Debugging

Every developer knows the benefit of debugging when a program is not working as intended. In OWL, the reasoner can give explanations to why an ontology is inconsistent. However, if your ontology is consistent but semantically incorrect, the reasoner does not help.

The consistent maximal ABox finding algorithm might be of help when reviewing the semantics of an ontology. For example, when creating the use case ontology and subsequently the object modeling methodology, we went a bit back and forth with the models before we found the correct methodology. The author struggled with modeling the locations correctly. Specifically, the TBox modeled allowed ABoxes which were not concrete worlds. Thus there was a bug in the TBox, and this was discovered by running the algorithm to compute all of the maximal ABoxes.

The maximal ABoxes let us see if the TBox was too restrictive or too lenient. If sets were generated that we did not wish to be generated, we had made a mistake in the modeling of allowing too much. While, if sets were not generated that we wished to be generated, we were too restrictive in the modeling. In our case it was too lenient, as it allowed ABoxes which we wanted to be inconsistent.

The specific problem was the following. At first, we modeled the locations as shown in Figure 7.3. This TBox created the issue allowing ABoxes to be stored where not all the individuals were assigned one of L, LF and BP. Assume an ontology with individuals a and b. Then the following ABox would be considered maximal and consistent.

$$\{GeoUnit(a), GeoUnit(b), frontOf(a,b), BP(b), Porous(a), NonPorous(b)\}$$

$$
\begin{array}{|l|}
\hline
L \sqsubseteq GeoUnit \\
LF \sqsubseteq GeoUnit \\
BP \sqsubseteq GeoUnit \\
L \sqsubseteq \forall frontOf.L \sqcup \forall frontOf.LF \\
LF \sqsubseteq \forall frontOf.LF \sqcup \forall frontOf.BP \\
BP \sqsubseteq \forall frontOf.BP \\
L \sqcap NonPorous \sqsubseteq \bot \\
LF \sqcap Porous \sqsubseteq \bot \\
BP \sqcap Porous \sqsubseteq \bot \\
\hline
\end{array}
$$

Figure 7.3: Problem TBox

The problem is that there is no location a is allowed to be in. If it is in front of something located inside of BP, it must be located in LF or BP. However, something which is located inside of LF or BP cannot also be $Porous$. Because a is not assigned any location, it is not a concrete world and thus should not be a maximal ABox. Further, assigning something located in front of BP to be $Porous$ should be illegal, and thus inconsistent. The problem with the TBox is that it is consistent even when an individual which is a $GeoUnit$ is not one of FC, DC or L. The solution we came up with was to force assignment of variables by using equivalence and disjoint union as was presented in Section 5.1.1 on page 46 on object modeling in OWL.

Using the algorithm to test the ontology this way might be useful in more cases than just when generating concrete worlds to find mistakes with the semantics of the ontology. However, we do not explore it further in this thesis.

7.5 Summary

When it comes to the questions posed in the beginning of the chapter, we conclude that the new approach does well on 3 out of the 4 evaluation questions when compared with the intertwined approach. It is a decent tool to use for end-users if they are somewhat knowledgeable of Description Logic or willing to learn it. For the developer the modularity of the system makes it easier to maintain and expand upon. The efficiency is a point of concern. In particular the reliance on an expressiveness which makes the reasoner N2EXPTIME-complete. Lastly, the results generated are correct as long as the ontology is semantically correct.

To sum up, the approach works and it makes use of a knowledge representation framework to simplify the formalization of domain knowledge. It has some issues with its efficiency, but, as we discussed, there are several ways to optimize the runtime to mitigate this to some extent.

Chapter 8

Conclusions

To end the thesis, we start by reviewing related work. Then we conclude by reviewing the goals of the thesis and end with describing topics which could be investigated further.

8.1 Related Work

The biggest contribution of this thesis is the methodology to solve a type of combinatorial problem with the help of OWL ontologies. Thus, the related work is mostly other approaches that can be used to solve the same or similar problems. Configuration design [45] is a similar type of problem to the concretization problem. In fact, the way worlds are formalized in Chapter 4 is likely a specific case of configuration design as configuration design involves components, relations, constraints and requirements. However, there is one big difference, the goal of configuration solving is to find one (potentially) optimal solution rather than all solutions.

The work closest to this thesis that we are aware of is the knowledge representation part of [6] by Din et al. where they work on the whole pipeline of the Geological Assistant. Din et al. use the logic programming language Prolog for the scenario pre-processing in the Geological Assistant, and this entails the following differences. First of all, in Prolog one has some algorithmic control such as the foreach predicate. Further, Prolog is based on the Turing complete Horn Logic, which is a specific way of formalizing axioms. Prolog is thus constrained to facts (assertions) and implications. On the other hand, in DL there is a larger variety of axioms available. As DL is purely focused on knowledge representation, it arguably makes formalization of knowledge easier as there are more axioms available and one does not have to think about the algorithmic control. However, Prolog is more applicable when a DL is not expressive enough for the domain modeled, for example if it is necessary with ter-

tiary relations. Furthermore, there are more possibilities to optimize a computation in Prolog as one has some algorithmic control. As DL is a pure modeling language, the models are easier to read and understand which is a benefit if the model is supposed to be shared and understood by various people.

In [42] a form of declarative (logic) programming, later named answer set programming, is proposed and applied on configuration design. It is as Prolog based on Horn logic, but it differs from Prolog in that one has no algorithmic control and stable model semantics are used.

Constraint programming is a declarative programming paradigm which is also used for combinatorial problems. It mainly consists of (decision) variables and constraints (relations). A constraint solver takes real-world problems in terms of variables and constraints as inputs, and finds an assignment to all the variables that satisfies the constraints [12]. E.g. a configuration design problem such as in [18], where constraint programming is combined with DL. If there is a constraint solver which is optimized to find all satisfying assignments, this could be an interesting tool for the Geological Assistant project. The differences between the two approaches would then be how the knowledge is formalized (DL vs. constraints) and how the computation is done (reasoner + Algorithm 2 on page 53 vs. constraint solver).

8.2 Conclusions and Future Work

In this thesis we have formalized the notion of underdetermined and concrete worlds, and shown an approach of representing them with OWL ontologies. An algorithm was developed which uses an OWL ontology to generate all of the concretizations of an underdetermined world. Then, by using Maude, an approach for converting concrete worlds serialized from an OWL ABox to a Maude configuration was presented. Before ending the thesis with proposing topics for future work, we now conclude by reviewing how the research questions RQ1, RQ2 and RQ3 from Section 1.3 on page 5 were answered.

RQ1 was on the topic of how underdetermined worlds can be represented within OWL. To help answer this, we first made an abstract definition of what worlds are in Chapter 4. Then we showed how the worlds can be represented with OWL by using various types of OWL axioms in Section 5.1.1. In particular, value sets were represented with disjoint unions of classes. We did not formalize how object dependencies could be represented as it depends on the domain of the knowledge being formalized. However, we made an OWL ontology of the use case presented in Sections 3.3 and 3.4 where we showed how one type of object dependency could be represented with value restrictions.

RQ2 regarded the topic of the concretizations of underdetermined worlds by using OWL. For this, we developed and presented an algorithm in Section 5.2.3

which uses the OWL API to concretize underdetermined worlds modeled with OWL ontologies. The algorithm generated all of the largest consistent combinations of class assertions of an ontology. We argued for the correctness of the algorithm in Section 5.2.5 through a proof that showed it generates sound results. Further, we partly proved that it generates complete results and gave an intuition for the part not proven. Lastly, we successfully applied the algorithm on the examples shown in Chapter 4 and a modification of the petroleum system use case presented in Sections 3.3 and 3.4.

RQ3 was about converting a representation of worlds in OWL into a representation of worlds in Maude. This was answered in Section 6.2 by showing how the OWL ABox entities could be mapped to entities in Maude by referring to the concepts of the abstract world definitions. Then, in Section 6.3, we presented an implementation of a conversion done in Maude.

Future Work

There are several aspects that would be interesting to investigate further. First of all, although the intuition of the part of the completeness proof considering generation of all combinations was explained, we did not formally prove it. Thus this part of the proof remains to be done. One of the more interesting topics to research is to investigate whether the consistent maximal ABox algorithm developed is applicable with already existing ontologies that do not use the modeling approach presented in this thesis. Further, if it is not applicable, whether there are steps that can be taken to make it applicable. Another interesting topic is to enable on-the-fly communication between Maude and OWL. Conceivably, this could be achieved by translating a fragment of a Maude configuration which is expressable in OWL to an OWL ABox. It would require that the TBox and RBox of the OWL ontology corresponds to the Maude declarations of the fragment. It would also be interesting to evaluate how much the impact is of the different suggested optimizations in Section 7.3 on page 86. Further reviewing of the applicability of the consistent maximal ABox algorithm for debugging semantics of ontologies would also be interesting. For the specific use case posed in this thesis work, it would be beneficial to find a modeling approach which does not require the direct semantics of OWL 2 as the computational complexity would be greatly improved.

Appendix A

Figures

This appendix presents figures not shown in the text. Figure A.1 shows the rest of the use case ontology and extends upon Figure 6.2 on page 67. Figure A.2 shows the actual TBox of the ontology used to generate the test results in Section 7.2.2 on page 85. Figure A.3 shows all of the possible orderings of 4 geological units given the formalization in Figure 3.6 on page 34.

Permeability and Porosity Axioms

$FC \sqsubseteq \neg NonPermeable \sqcap \neg NonPorous$
$IC_1 \sqsubseteq \neg Permeable \sqcap \neg Porous$
$DC \sqsubseteq \neg NonPermeable \sqcap \neg NonPorous$
$IC_2 \sqsubseteq \neg Permeable \sqcap \neg Porous$
$L \sqsubseteq \neg NonPermeable \sqcap \neg NonPorous$
$LF \sqsubseteq \neg Permeable \sqcap \neg Porous$
$BP \sqsubseteq \neg Permeable \sqcap \neg Porous$

Value Set Disjointness Axioms

$Sealing \sqcap NonSealing \sqsubseteq \bot$	$IC_1 \sqcap LF \sqsubseteq \bot$
$Permeable \sqcap NonPermeable \sqsubseteq \bot$	$IC_1 \sqcap BP \sqsubseteq \bot$
$Porous \sqcap NonPorous \sqsubseteq \bot$	$DC \sqcap IC_2 \sqsubseteq \bot$
$FC \sqcap IC_1 \sqsubseteq \bot$	$DC \sqcap L \sqsubseteq \bot$
$FC \sqcap DC \sqsubseteq \bot$	$DC \sqcap LF \sqsubseteq \bot$
$FC \sqcap IC_2 \sqsubseteq \bot$	$DC \sqcap BP \sqsubseteq \bot$
$FC \sqcap L \sqsubseteq \bot$	$IC_2 \sqcap L \sqsubseteq \bot$
$FC \sqcap LF \sqsubseteq \bot$	$IC_2 \sqcap LF \sqsubseteq \bot$
$FC \sqcap BP \sqsubseteq \bot$	$IC_2 \sqcap BP \sqsubseteq \bot$
$IC_1 \sqcap DC \sqsubseteq \bot$	$L \sqcap LF \sqsubseteq \bot$
$IC_1 \sqcap IC_2 \sqsubseteq \bot$	$L \sqcap BP \sqsubseteq \bot$
$IC_1 \sqcap L \sqsubseteq \bot$	$LF \sqcap BP \sqsubseteq \bot$

Figure A.1: Rest of use case TBox $\mathcal{T}_U$

GeoUnit Object Axioms

$GeoUnit \equiv Location$
$GeoUnit \sqsubseteq \neg SealingCapacity$
$Fault \equiv SealingCapacity$
$Fault \sqsubseteq \neg Location$

Value Set Axioms

$SealingCapacity \equiv Sealing \sqcup NonSealing$
$Location \equiv FC \sqcup IC_1 \sqcup DC \sqcup IC_2 \sqcup L \sqcup LF \sqcup BP$

Dependency Axioms

$FC \sqsubseteq \forall frontOf.FC \sqcup \forall frontOf.IC_1 \sqcup \forall frontOf.DC$
$IC_1 \sqsubseteq \forall frontOf.IC_1 \sqcup \forall frontOf.DC$
$DC \sqsubseteq \forall frontOf.DC \sqcup \forall frontOf.IC_2 \sqcup \forall frontOf.L$
$IC_2 \sqsubseteq \forall frontOf.IC_2 \sqcup \forall frontOf.L$
$L \sqsubseteq \forall frontOf.L \sqcup \forall frontOf.LF$
$LF \sqsubseteq \forall frontOf.LF \sqcup \forall frontOf.BP$
$BP \sqsubseteq \forall frontOf.BP$

Object Property Axioms

$\top \sqsubseteq \leq 1\ frontOf\ Thing$ (functional)
$\top \sqsubseteq \leq 1\ frontOf^{-}\ Thing$ (inverse functional)
$\top \sqsubseteq \neg \exists frontOf.Self$ (irreflexive)

Value Set Disjointness Axioms

$Sealing \sqcap NonSealing \sqsubseteq \bot$
$FC \sqcap IC_1 \sqsubseteq \bot$
$FC \sqcap DC \sqsubseteq \bot$
$FC \sqcap IC_2 \sqsubseteq \bot$
$FC \sqcap L \sqsubseteq \bot$
$FC \sqcap LF \sqsubseteq \bot$
$FC \sqcap BP \sqsubseteq \bot$
$IC_1 \sqcap DC \sqsubseteq \bot$
$IC_1 \sqcap IC_2 \sqsubseteq \bot$
$IC_1 \sqcap L \sqsubseteq \bot$
$IC_1 \sqcap LF \sqsubseteq \bot$
$DC \sqcap L \sqsubseteq \bot$
$DC \sqcap LF \sqsubseteq \bot$
$DC \sqcap BP \sqsubseteq \bot$
$IC_2 \sqcap L \sqsubseteq \bot$
$IC_2 \sqcap LF \sqsubseteq \bot$
$IC_2 \sqcap BP \sqsubseteq \bot$
$L \sqcap LF \sqsubseteq \bot$
$L \sqcap BP \sqsubseteq \bot$
$LF \sqcap BP \sqsubseteq \bot$

Figure A.2: Actual TBox used in the testing

1. $FC \to FC \to FC \to FC$	24. $IC_1 \to IC_1 \to DC \to IC_2$	47. $DC \to L \to LF \to LF$
2. $FC \to FC \to FC \to IC_1$	25. $IC_1 \to IC_1 \to DC \to L$	48. $IC_2 \to IC_2 \to IC_2 \to IC_2$
3. $FC \to FC \to FC \to DC$	26. $IC_1 \to DC \to DC \to DC$	49. $IC_2 \to IC_2 \to IC_2 \to L$
4. $FC \to FC \to IC_1 \to IC_1$	27. $IC_1 \to DC \to DC \to IC_2$	50. $IC_2 \to IC_2 \to L \to L$
5. $FC \to FC \to IC_1 \to DC$	28. $IC_1 \to DC \to DC \to L$	51. $IC_2 \to IC_2 \to L \to LF$
6. $FC \to FC \to DC \to DC$	29. $IC_1 \to DC \to IC_2 \to IC_2$	52. $IC_2 \to L \to L \to L$
7. $FC \to FC \to DC \to IC_2$	30. $IC_1 \to DC \to IC_2 \to L$	53. $IC_2 \to L \to L \to LF$
8. $FC \to FC \to DC \to L$	31. $IC_1 \to DC \to DC \to L$	54. $IC_2 \to L \to LF \to LF$
9. $FC \to IC_1 \to IC_1 \to IC_1$	32. $IC_1 \to DC \to L \to L$	55. $IC_2 \to L \to LF \to BP$
10. $FC \to IC_1 \to IC_1 \to DC$	33. $IC_1 \to DC \to L \to LF$	56. $L \to L \to L \to L$
11. $FC \to IC_1 \to DC \to DC$	34. $DC \to DC \to DC \to DC$	57. $L \to L \to L \to LF$
12. $FC \to IC_1 \to DC \to IC_2$	35. $DC \to DC \to DC \to IC_2$	58. $L \to L \to LF \to LF$
13. $FC \to IC_1 \to DC \to L$	36. $DC \to DC \to DC \to L$	59. $L \to L \to LF \to BP$
14. $FC \to DC \to DC \to DC$	37. $DC \to DC \to IC_2 \to IC_2$	60. $L \to LF \to LF \to LF$
15. $FC \to DC \to DC \to IC_2$	38. $DC \to DC \to IC_2 \to L$	61. $L \to LF \to LF \to BP$
16. $FC \to DC \to DC \to L$	39. $DC \to DC \to L \to L$	62. $L \to LF \to BP \to BP$
17. $FC \to DC \to IC_2 \to IC_2$	40. $DC \to DC \to L \to LF$	63. $LF \to LF \to LF \to LF$
18. $FC \to DC \to IC_2 \to L$	41. $DC \to IC_2 \to IC_2 \to IC_2$	64. $LF \to LF \to LF \to BP$
19. $FC \to DC \to L \to L$	42. $DC \to IC_2 \to IC_2 \to L$	65. $LF \to LF \to BP \to BP$
20. $FC \to DC \to L \to LF$	43. $DC \to IC_2 \to L \to L$	66. $LF \to BP \to BP \to BP$
21. $IC_1 \to IC_1 \to IC_1 \to IC_1$	44. $DC \to IC_2 \to L \to LF$	67. $BP \to BP \to BP \to BP$
22. $IC_1 \to IC_1 \to IC_1 \to DC$	45. $DC \to L \to L \to L$	
23. $IC_1 \to IC_1 \to DC \to DC$	46. $DC \to L \to L \to LF$	

Figure A.3: Location orders

Appendix B

Complete Syntax Converter Example

In this appendix we present the complete Maude module from the example in Section 6.3 on page 69.

Listing B.1: Complete Maude module of the syntax converter from Section 6.3

```
mod OWL-CONVERTER is
  protecting CONFIGURATION .
  PROTECTING STRING .

  sorts OWLLocation OWLPermeability OWLPorosity .
  subsort OWLLocation OWLPermeability OWLPorosity < Value .

  --- values
  op FeederChannel : -> OWLLocation .
  op Permeable : -> OWLPermeability .
  op Porous : -> OWLPorosity .

  --- object names and value sets
  op GeoUnit : -> ObjectName .
  op Location : -> ValueSet .
  op Permeability : -> ValueSet .
  op Porosity : -> ValueSet .

  sorts ObjectAttribute LocationT PermeabilityT PorosityT .
  subsorts LocationT PermeabilityT PorosityT < ObjectAttribute .

  op feederChannel : -> LocationT .
  op permeable : -> PermeabilityT .
  op porous : -> PorosityT .
```

```
  --- operator to construct maude objects from an OWL message
  op type : OWLClass Oid -> Msg .

  op <_: GeoUnit | Permeability:_, Porosity:_, Location:_> :
     Oid PermeabilityT PorosityT LocationT -> Object [ctor] .

  --- conversion function
  op convert : Value -> ObjectAttribute .
  eq convert(FeederChannel) = feederChannel .
  eq convert(Permeable) = permeable .
  eq convert(Porous) = porous .

  var O : Oid .
  var PB : OWLPermeability .
  var PR : OWLPorosity .
  var L : OWLLocation .

  eq type(GeoUnit, O) type(Permeability, O) type(PB, O)
  type(Porosity, O type(PR, O) type(Location, O) type(L, O)
    =
      < O : GeoUnit | Permeability: convert(PB), Porosity: convert(PR),
         Location: convert(L) > .
endm
```

Appendix C

Implementation of the Consistent Maximal ABox Algorithm

In this appendix we present code from our implementation of Algorithm 2 on page 53.[1] Most comments are removed from the listings, please see the full implementation on Github if you are interested in them. The implementation imports packages from the OWL API. For more information on the OWL API, see its wiki[2] and Javadocs[3]. The implementation on Github uses Maven[4] for building and its package dependencies which are the OWL API[5] and the HermiT Reasoner[6].

Listing C.1 shows the direct implementation of Algorithm 2, while the rest of the listings show code used by the direct implementation. Listing C.2 shows a class wrapping the OWLOntology class from the OWL API, Listing C.3 shows utility methods (in particular the combination subset test in the base case of the algorithm) and Listing C.4 shows the method serializing ontologies.

[1] `https://github.com/vskaret/consistent-maximal-abox-generator`
[2] `https://github.com/owlcs/owlapi/wiki`
[3] `http://owlcs.github.io/owlapi/apidocs_4/index.html`
[4] https://maven.apache.org/
[5] `https://mvnrepository.com/artifact/net.sourceforge.owlapi`
[6] `https://mvnrepository.com/artifact/net.sourceforge.owlapi/org.semanticweb.hermit`

Listing C.1: Implemenation of the consistent maximal ABox algorithm

```
1  package com.geoassistant.scenariogen;
2  import org.semanticweb.owlapi.model.*;
3  import java.util.*;
4
5  public class OntologyPermuter extends Ontology {
6      String unknownClassName = "Unknown";
7      String nonPermutable = "NonPermutable";
8      String maudePath = "src/maude/";
9      String outputPath = maudePath + "combinations/proto-scenario";
10     int combinationCounter = 0;
11     Set<Set<OWLClassAssertionAxiom>> combinations = new HashSet<>();
12     Set<OWLClassAssertionAxiom> combination = new HashSet<>();
13
14     public OntologyPermuter() {}
15
16     /**
17      * Initial call to start the permutation process. This process
18      * generates new ontologies based on the rules in the ontology.
19      */
20     public void permute() throws Exception {
21         OWLClass thing = factory.getOWLThing();
22         Set<OWLClassAssertionAxiom> unknownSet = getClassAssertionAxioms
               (unknownClassName);
23         OWLClassAssertionAxiom[] unknownArr = unknownSet.toArray(new
               OWLClassAssertionAxiom[unknownSet.size()]);
24         ArrayList<OWLClassAssertionAxiom> permutables = new ArrayList<>(
               Arrays.asList(unknownArr));
25
26         if (!permutables.isEmpty()) {
27             permute(permutables, thing);
28         }
29     }
30
31     /**
32      * Intermediary permuting call. Used to add children of owl:Thing.
33      */
34     public void permute(ArrayList<OWLClassAssertionAxiom> permutables,
           OWLClass root) throws Exception {
35         Queue<OWLClass> queue = new LinkedList<>();
```

```
36          OWLClassAssertionAxiom unknownCAA = permutables.get(0);
37          permutables = Utils.copyWithoutFirstElement(permutables);
38          OWLIndividual unknown = unknownCAA.getIndividual();
39          manager.removeAxiom(ont, unknownCAA);
40
41          for (OWLClass c : reasoner.getSubClasses(root, true).
                getFlattened()) {
42              if (!c.getIRI().getShortForm().equals(unknownClassName) && !c
                    .getIRI().getShortForm().equals(nonPermutable)) {
43                  queue.add(c);
44              }
45          }
46          bfs(permutables, queue, unknown);
47      }
48
49      /**
50       * Recursive breadth-first traversal of the class hierarchy for an
51       * individual (named treeTraverse in the pseudo code)
52       */
53      private void bfs(ArrayList<OWLClassAssertionAxiom> permutables,
            Queue<OWLClass> queue, OWLIndividual unknown) throws Exception {
54          if (queue.isEmpty() && permutables.isEmpty()) {
55              if (!Utils.subsetOf(combination, combinations)) {
56                  System.out.print(++combinationCounter + " ");
57                  OntologyUtils.prettyprint(combination);
58                  MaudeSerializer.writeFile(ont, reasoner, outputPath +
                        combinationCounter + ".maude");
59
60                  Set<OWLClassAssertionAxiom> res = new HashSet<>(
                        combination);
61                  combinations.add(res);
62              }
63              return;
64          } else if (queue.isEmpty() && !permutables.isEmpty()) {
65              permute(permutables, factory.getOWLThing());
66              return;
67          }
68
69          OWLClass currentClass = queue.poll();
70          Queue<OWLClass> queueCopy = new LinkedList<>(queue);
```

```
71          OWLClassAssertionAxiom ax = factory.getOWLClassAssertionAxiom(
                currentClass, unknown);
72          boolean containedAxiom = ont.containsAxiom(ax);
73
74          if (!containedAxiom) {
75              manager.addAxiom(ont, ax);
76              reasoner.flush();
77          }
78
79          if (reasoner.isConsistent()) {
80              combination.add(ax);
81
82              for (OWLClass c : reasoner.getSubClasses(currentClass, true).
                    getFlattened()) {
83                  if (!c.isOWLNothing()) {
84                      queue.add(c);
85                  }
86              }
87
88              bfs(permutables, queue, unknown);
89              combination.remove(ax);
90          }
91
92          if (!containedAxiom) {
93              manager.removeAxiom(ont, ax);
94              reasoner.flush();
95          }
96
97          bfs(permutables, queueCopy, unknown);
98      }
99  }
```

Listing C.2: The class Ontology which wraps around the OWLOntology class

```
1  package com.geoassistant.scenariogen;
2  import org.semanticweb.HermiT.Reasoner;
3  import org.semanticweb.owlapi.apibinding.OWLManager;
4  import org.semanticweb.owlapi.model.*;
5  import org.semanticweb.owlapi.reasoner.OWLReasoner;
6  import org.semanticweb.owlapi.vocab.PrefixOWLOntologyFormat;
7  import java.io.File;
```

```
 8 import java.util.*;
 9
10 /**
11  * This abstract class is a representation of an ontology in the OWLAPI
        .
12  * Each ontology will has its own Ontology Manager.
13  */
14 public abstract class Ontology {
15     File ontFile;
16     final String srcPath = System.getProperty("user.dir") + "/src/";
17     final String cwd = srcPath + "main/java/com/geoassistant/scenariogen
           /";
18     final String owldir = srcPath + "owl/";
19     OWLOntologyManager manager;
20     OWLOntology ont;
21     OWLReasoner reasoner;
22     OWLDataFactory factory;
23     PrefixOWLOntologyFormat pm;
24     String ontIRI;
25
26     public Ontology() {
27         manager = OWLManager.createOWLOntologyManager();
28     }
29
30     /**
31      * Load an ontology from a file from same folder as the java source
            file. Also instantiates a Reasoner object.
32      */
33     protected void loadOntology(String filename) throws Exception {
34         if (ont != null) {
35             throw new Exception("Already loaded an ontology.");
36         }
37
38         this.ontFile = new File(owldir + filename);
39
40         ont = manager.loadOntologyFromOntologyDocument(ontologyFile);
41         reasoner = new Reasoner.ReasonerFactory().createReasoner(
               ontology);
42
43         if (!reasoner.isConsistent()) {
```

```
44              throw new Exception("Loaded ontology is not consistent.");
45          }
46
47          factory = manager.getOWLDataFactory();
48          pm = (PrefixOWLOntologyFormat) manager.getOntologyFormat(ont);
49      }
50
51      /**
52       * Get all class assertion axioms concerning className
53       * (all individuals instantiated to be this class).
54       */
55      public Set<OWLClassAssertionAxiom> getClassAssertionAxioms(String
            className) {
56          OWLClassExpression oce = factory.getOWLClass(className, pm);
57          return ont.getClassAssertionAxioms(oce);
58      }
59  }
```

Listing C.3: Utility methods

```
1   package com.geoassistant.scenariogen;
2   import java.util.ArrayList;
3   import java.util.Set;
4
5   public final class Utils {
6       /**
7        * Checks if a set is a sub of a set of sets.
8        */
9       public static <T> boolean subsetOf(Set<T> set, Set<Set<T>> mainSet)
            {
10          if (mainSet.isEmpty()) {
11              return false;
12          }
13
14          for (Set<T> s : mainSet) {
15              if (s.containsAll(set)) {
16                  return true;
17              }
18          }
19
20          return false;
```

```
21      }
22
23      /**
24       * Returns a copy of a list without its first element.
25       */
26      public static <T> ArrayList<T> copyWithoutFirstElement(ArrayList<T>
            list) throws Exception {
27          if (list.isEmpty()) {
28              throw new Exception("can't remove from empty list");
29          }
30
31          ArrayList<T> copy = new ArrayList<>(list);
32          copy.remove(0);
33          return copy;
34      }
35  }
```

Listing C.4: MaudeSerializer. The class that serializes an OWLOntology into a Maude configuration as depicted in Listing 6.8 on page 72

```
1   package com.geoassistant.scenariogen;
2   import org.semanticweb.owlapi.model.*;
3   import org.semanticweb.owlapi.reasoner.OWLReasoner;
4   import java.io.IOException;
5   import java.io.*;
6   import java.util.Set;
7
8   public class MaudeSerializer {
9       static final String prefix = "mod OWL-ABOX is\n" +
10              " protecting OWL-CONVERTER .\n";
11      static final String postfix = "\nendm";
12
13      public static String serializeAbox(OWLOntology ont, OWLReasoner
            reasoner) {
14          Set<OWLNamedIndividual> individuals = ont.
                getIndividualsInSignature();
15          StringBuffer res = new StringBuffer(prefix);
16          res.append(" op unknowns : -> Configuration .\n");
17          res.append(" eq unknowns = ");
18
19          for (OWLNamedIndividual i : individuals) {
```

```
20              String oid = i.getIRI().getShortForm();
21              Set<OWLClassAssertionAxiom> instantiations = ont.
                    getClassAssertionAxioms(i);
22
23              for (OWLClassAssertionAxiom cax : instantiations) {
24                  String sort = cax.getClassExpression().asOWLClass().
                        getIRI().getShortForm();
25                  res.append("type(" + sort + ", " + '"' + oid + '"' + ") "
                        );
26              }
27
28              // object properties
29              Set<OWLObjectPropertyAssertionAxiom> axioms = ont.
                    getObjectPropertyAssertionAxioms(i);
30              for (OWLObjectPropertyAssertionAxiom ax : axioms) {
31                  String prop = ax.getProperty().asOWLObjectProperty().
                        getIRI().getShortForm();
32                  String subject = ax.getSubject().asOWLNamedIndividual().
                        getIRI().getShortForm();
33                  String object = ax.getObject().asOWLNamedIndividual().
                        getIRI().getShortForm();
34
35                  res.append(prop + "(\"" + subject + "\", \"" + object + "
                        \") ");
36              }
37          }
38          res.append(".");
39          res.append(postfix);
40          return res.toString();
41      }
42
43      public static void writeFile(OWLOntology ont, OWLReasoner reasoner,
            String filename) throws IOException {
44          String output = serializeAbox(ont, reasoner);
45          BufferedWriter writer = new BufferedWriter(new FileWriter(
                filename));
46          writer.write(output);
47          writer.close();
48      }
49  }
```

Bibliography

[1] Joseph L Allen. "Faults, Frictional Melts, and Fossil Fuel Reservoirs: NSF and PRF Projects Supporting Undergraduate Research at Concord University". In: *Proceedings of STaR Symposium 2009*. 2009, pp. 54–61.

[2] Franz Baader, Sebastian Brandt, and Carsten Lutz. "Pushing the EL Envelope". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, 2005, pp. 364–369. URL: http://ijcai.org/Proceedings/05/Papers/0372.pdf.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, eds. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. ISBN: 0-521-78176-0.

[4] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. "Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family". In: *J. Autom. Reasoning* 39.3 (2007), pp. 385–429. DOI: 10.1007/s10817-007-9078-x. URL: https://doi.org/10.1007/s10817-007-9078-x.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. "Maude: specification and programming in rewriting logic". In: *Theor. Comput. Sci.* 285.2 (2002), pp. 187–243. DOI: 10.1016/S0304-3975(01)00359-0. URL: https://doi.org/10.1016/S0304-3975(01)00359-0.

[6] Crystal Chang Din, Leif Harald Karlsen, Irina Pene, Oliver Stahl, Ingrid Chieh Yu, and Thomas Østerlie. "Geological Multi-scenario Reasoning". In: *32nd Norsk Informatikkonferanse, NIK 2019, UIT Norges Arktiske Universitet, Narvik, Norway, November 25-27, 2019*. Bibsys Open Journal Systems, Norway, 2019. URL: https://ojs.bibsys.no/index.php/NIK/article/view/640.

[7] Renee Elio, Jim Hoover, Ioanis Nikolaidis, Mohammad Salavatipour, Lorna Stewart, and Ken Wong. *About computing science research methodology*. 2011.

[8] Michael J. Fischer and Richard E. Ladner. "Propositional Dynamic Logic of Regular Programs". In: *J. Comput. Syst. Sci.* 18.2 (1979), pp. 194–211. DOI: 10.1016/0022-0000(79)90046-1. URL: https://doi.org/10.1016/0022-0000(79)90046-1.

[9] *GEO ExPro Seismic Image*. [Online; accessed: 31-May-2019]. URL: https://www.geoexpro.com/articles/2016/01/super-high-resolution-seismic-data-in-the-norwegian-barents-sea.

[10] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. "HermiT: An OWL 2 Reasoner". In: *J. Autom. Reasoning* 53.3 (2014), pp. 245–269. DOI: 10.1007/s10817-014-9305-1. URL: https://doi.org/10.1007/s10817-014-9305-1.

[11] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter F. Patel-Schneider, and Ulrike Sattler. "OWL 2: The next step for OWL". In: *J. Web Semant.* 6.4 (2008), pp. 309–322. DOI: 10.1016/j.websem.2008.05.001. URL: https://doi.org/10.1016/j.websem.2008.05.001.

[12] Frank van Harmelen, Vladimir Lifschitz, and Bruce W. Porter, eds. *Handbook of Knowledge Representation*. Vol. 3. Foundations of Artificial Intelligence. Elsevier, 2008. ISBN: 978-0-444-52211-5. URL: http://www.sciencedirect.com/science/bookseries/15746526/3.

[13] Robert Hoehndorf, Paul N. Schofield, and Georgios V. Gkoutos. "The role of ontologies in biological and biomedical research: a functional perspective". In: *Briefings Bioinform.* 16.6 (2015), pp. 1069–1080. DOI: 10.1093/bib/bbv011. URL: https://doi.org/10.1093/bib/bbv011.

[14] Matthew Horridge and Sean Bechhofer. "The OWL API: A Java API for Working with OWL 2 Ontologies". In: *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009*. Ed. by Rinke Hoekstra and Peter F. Patel-Schneider. Vol. 529. CEUR Workshop Proceedings. CEUR-WS.org, 2009. URL: http://ceur-ws.org/Vol-529/owled2009%5C_submission%5C_29.pdf.

[15] Matthew Horridge, Nick Drummond, John Goodwin, Alan L. Rector, Robert Stevens, and Hai Wang. "The Manchester OWL Syntax". In: *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006*. Ed. by Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace. Vol. 216. CEUR Workshop Proceedings. CEUR-WS.org, 2006. URL: http://ceur-ws.org/Vol-216/submission%5C_9.pdf.

[16] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. "The Even More Irresistible SROIQ". In: (2006). Ed. by Patrick Doherty, John Mylopoulos, and Christopher A. Welty, pp. 57–67. URL: http://www.aaai.org/Library/KR/2006/kr06-009.php.

[17] John M Hunt. *Petroleum geochemistry and geology*. eng. A Series of books in geology. San Francisco: W. H. Freeman, 1979. ISBN: 0716710056.

[18] Ulrich Junker and Daniel Mailharro. "The logic of ilog (j) configurator: Combining constraint programming with a description logic". In: *proceedings of Workshop on Configuration, IJCAI*. Vol. 3. Citeseer. 2003, pp. 13–20.

[19] Roman Kontchakov, Martin Rezk, Mariano Rodriguez-Muro, Guohui Xiao, and Michael Zakharyaschev. "Answering SPARQL Queries over Databases under OWL 2 QL Entailment Regime". In: *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part I*. Ed. by Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul T. Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble. Vol. 8796. Lecture Notes in Computer Science. Springer, 2014, pp. 552–567. DOI: 10.1007/978-3-319-11964-9_35. URL: https://doi.org/10.1007/978-3-319-11964-9%5C_35.

[20] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. "A Description Logic Primer". In: *CoRR* abs/1201.4089 (2012). arXiv: 1201.4089. URL: http://arxiv.org/abs/1201.4089.

[21] A. I Levorsen. *Geology of petroleum*. eng. 2nd ed. A Series of books in geology. San Francisco, Calif: W.H. Freeman, 1967.

[22] Christopher L Liner. *The Art and Science of Seismic Interpretation*. eng. 1st ed. 2019. SpringerBriefs in Earth Sciences. Cham: Springer International Publishing : Imprint: Springer, 2019. ISBN: 3-030-03998-6.

[23] LB Magoon and EA Beaumont. "Petroleum system (chapter 3)". In: *Exploring for Oil and Gas Traps, Edward A. Beaumont and Norman H. Foster, eds., Treatise of Petroleum Geology, Handbook of Petroleum Geology* (1999), p. 34.

[24] Leslie B. Magoon and Wallace G. Dow. "The Petroleum System". In: *The Petroleum System—From Source to Trap*. American Association of Petroleum Geologists, Jan. 1994. ISBN: 9781629810928. DOI: 10.1306/M60585C1. URL: https://doi.org/10.1306/M60585C1.

[25] Nicolas Matentzoglu, Jared Leo, Valentino Hudhra, Uli Sattler, and Bijan Parsia. "A Survey of Current, Stand-alone OWL Reasoners". In: *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description Logics (DL 2015), Athens, Greece, June 6, 2015*. Ed. by Michel Dumontier, Birte Glimm, Rafael S.

Gonçalves, Matthew Horridge, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Giorgos B. Stamou, and Giorgos Stoilos. Vol. 1387. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 68–79. URL: http://ceur-ws.org/Vol-1387/paper%5C_4.pdf.

[26] Deborah L McGuinness, Frank Van Harmelen, et al. "OWL web ontology language overview". In: *W3C recommendation* 10.10 (2004), p. 2004.

[27] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, Carsten Lutz, et al. "OWL 2 web ontology language profiles". In: *W3C recommendation* 27 (2009), p. 61.

[28] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. "Parallel Materialisation of Datalog Programs in Centralised, Main-Memory RDF Systems". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, 2014, pp. 129–137. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8505.

[29] Mark A Musen. "The protégé project: a look back and a look forward". In: *AI matters* 1.4 (2015), pp. 4–12.

[30] E. Mutti and F. Ricci Lucchi. "Turbidites of the northern Apennines: introduction to facies analysis". In: *International Geology Review* 20.2 (1978), pp. 125–166. DOI: 10.1080/00206817809471524. eprint: https://doi.org/10.1080/00206817809471524. URL: https://doi.org/10.1080/00206817809471524.

[31] Peter C. Ölveczky. *Designing Reliable Distributed Systems. A Formal Methods Approach Based on Executable Modeling in Maude*. Springer-Verlag, 2017. ISBN: 978-1-4471-6687-0.

[32] David Lorge Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules". In: vol. 15. 12. 1972, pp. 1053–1058. DOI: 10.1145/361598.361623. URL: https://doi.org/10.1145/361598.361623.

[33] Peter F. Patel-Schneider. "System Description: DLP". In: *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*. Ed. by David A. McAllester. Vol. 1831. Lecture Notes in Computer Science. Springer, 2000, pp. 297–301. DOI: 10.1007/10721959_23. URL: https://doi.org/10.1007/10721959%5C_23.

[34] Sebastian Rudolph. "Foundations of Description Logics". In: *Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*. Ed. by Axel Polleres, Claudia d'Amato, Marcelo Arenas, Siegfried Handschuh, Paula Kroner, Sascha Ossowski, and Peter F. Patel-Schneider. Vol. 6848. Lecture Notes in

Computer Science. Springer, 2011, pp. 76–136. DOI: 10.1007/978-3-642-23032-5_2. URL: https://doi.org/10.1007/978-3-642-23032-5%5C_2.

[35] Gudula Rünger and Thomas Rauber. *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition*. Springer, 2013. ISBN: 978-3-642-37800-3. DOI: 10.1007/978-3-642-37801-0. URL: https://doi.org/10.1007/978-3-642-37801-0.

[36] *Science Direct*. [Online; accessed: 01-June-2019]. URL: https://www.sciencedirect.com/topics/earth-and-planetary-sciences/petroleum-system.

[37] G Shanmugam and R.J Moiola. "Submarine fans: Characteristics, models, classification, and reservoir potential". eng. In: vol. 24. 6. Elsevier B.V, 1988, pp. 383–428.

[38] G Shanmugam and RJ Moiola. "Submarine fan models: problems and solutions". In: *Submarine fans and related turbidite systems*. Frontiers in sedimentary geology. New York: Springer, 1985, pp. 29–35. ISBN: 3540961429.

[39] Michael Sipser. *Introduction to the theory of computation*. eng. 3rd ed. Australia: Cengage learning, 2013. ISBN: 9781133187813.

[40] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. "Pellet: A practical OWL-DL reasoner". In: *J. Web Semant.* 5.2 (2007), pp. 51–53. DOI: 10.1016/j.websem.2007.03.004. URL: https://doi.org/10.1016/j.websem.2007.03.004.

[41] Raymond M Smullyan. *First-order logic*. eng. Vol. Bd. 43. Ergebnisse der Mathematik und ihrer Grenzgebiete. Berlin: Springer, 1968. ISBN: 3540040994.

[42] Timo Soininen and Ilkka Niemelä. *Formalizing Configuration Knowledge Using Rules with Choices*. 1998.

[43] Pradeep Tharakan, John Hallock, Charles Hall, Michael Jefferson, and Cutler Cleveland. "Hydrocarbons and the evolution of human culture". eng. In: *Nature* 426.6964 (2003), pp. 318–322. ISSN: 0028-0836.

[44] Dmitry Tsarkov and Ian Horrocks. "FaCT++ Description Logic Reasoner: System Description". In: *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Ed. by Ulrich Furbach and Natarajan Shankar. Vol. 4130. Lecture Notes in Computer Science. Springer, 2006, pp. 292–297. DOI: 10.1007/11814771_26. URL: https://doi.org/10.1007/11814771%5C_26.

[45] Bob J. Wielinga and Guus Schreiber. "Configuration-Design Problem Solving". In: *IEEE Expert* 12.2 (1997), pp. 49–56. DOI: 10.1109/64.585104. URL: https://doi.org/10.1109/64.585104.

[46] Aliyu Yauri, Rabiah Abdul Kadir, Azreen Azman, and Masrah Murad. "Quranic Verse Extraction base on Concepts using OWL-DL Ontology". In: *Research Journal of Applied Sciences, Engineering and Technology* 6.23 (Dec. 2013), pp. 4492–4498. DOI: 10.19026/rjaset.6.3457.