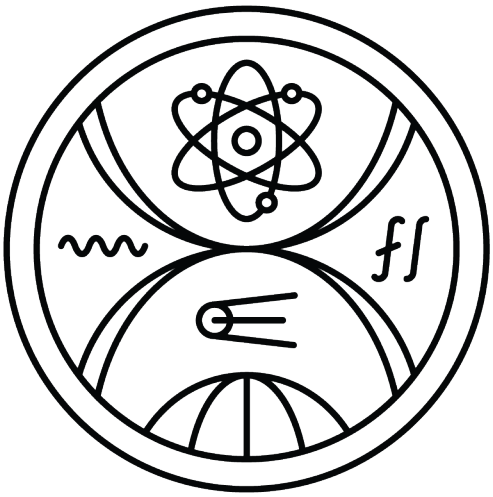
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND
INFORMATICS

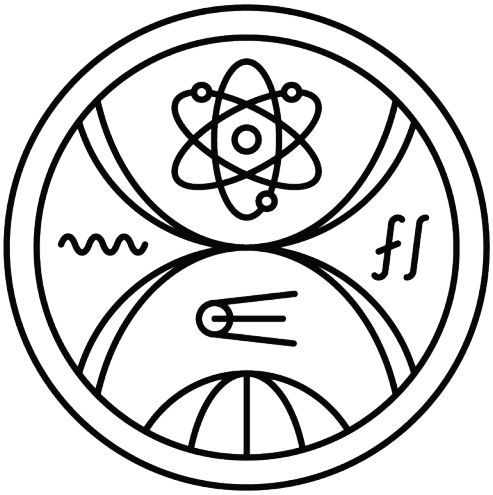


OPTIMIZATION OF THE MHS-MXP ALGORITHM

Master's thesis

2022 Bc. Janka Boborová

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS



OPTIMIZATION OF THE MHS-MXP ALGORITHM

Master's thesis

Study Program:	Applied Informatics
Field of Study:	2511 Applied Informatics
School Department:	Department of Applied Informatics
Supervisor:	Mgr. Júlia Pukancová, PhD.
Consultant:	doc. RNDr. Martin Homola, PhD.

Bratislava, 2022 Bc. Janka Boborová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Janka Boborová
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Optimization of the MHS-MXP algorithm

Annotation: The MHS-MXP abduction algorithm improves MHS by applying a divide-and-conquer strategy (MXP) to search through a space of possible explanations (MHS-tree). The MXP runs are iterated, therefore suitable search heuristics, tree-pruning, and cached information from previous runs may possibly further optimize the search strategy in the consecutive iterations.

Aim: Propose improvements in MHS-MXP search strategy, and evaluate their efficiency on a suitable test case.

Literature:
1. Elsenbroich, C., Kutz, O., Sattler, U., 2006. A case for abductive reasoning over ontologies. In: OWLED
2. Shchekotykhin, K., Jannach, D., Schmitz, T., 2015. MergeXplain: Fast computation of multiple conflicts for diagnosis. In IJCAI
3. Homola, M., Pukancová, J., Gablíková, J., Fabianová, K., 2020. Merge, Explain, Iterate. In: DL

Supervisor:	Mgr. Júlia Pukancová, PhD.
Consultant:	doc. RNDr. Martin Homola, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	prof. Ing. Igor Farkaš, Dr.
Assigned:	11.10.2021

Approved: 12.10.2021 prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Janka Boborová
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Optimization of the MHS-MXP algorithm
Optimalizácia algoritmu MHS-MXP

Anotácia: Abduktívny algoritmus MHS-MXP je zlepšením algoritmu MHS, ktoré využíva metódu rozdeľuj a panuj (MXP) pri prehľadávaní priestoru možných vysvetlení (MHS-strom). Behy MXP sú iterované, preto môže vhodná heuristika vyhľadávania, orezávanie stromu, a kešovanie informácií z predchádzajúcich behov potenciálne zlepšiť prehľadávaciu stratégiu v nasledujúcich iteráciách.

Cieľ: Navrhnúť možné zlepšenia prehľadávacej stratégie a evalvovať ich efektívnosť na vhodne zvolených testovacích dátach.

Literatúra:
1. Elsenbroich, C., Kutz, O., Sattler, U., 2006. A case for abductive reasoning over ontologies. In: OWLED
2. Shchekotykhin, K., Jannach, D., Schmitz, T., 2015. MergeXplain: Fast computation of multiple conflicts for diagnosis. In IJCAI
3. Homola, M., Pukancová, J., Gablíková, J., Fabianová, K., 2020. Merge, Explain, Iterate. In: DL

Vedúci: Mgr. Júlia Pukancová, PhD.
Konzultant: doc. RNDr. Martin Homola, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 11.10.2021

Dátum schválenia: 12.10.2021 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

I would like to express my deepest gratitude to my supervisor Júlia Pukancová and consultant Martin Homola for their support, valuable advice and willingness to answer all my questions. I am also grateful to my boyfriend who stood by my side even in difficult moments.

Abstract

Abduction is a type of inference whose task is to explain an observation based on the background knowledge we have. Several algorithms have been proposed to solve the abduction problem. We deal with the standard MHS algorithm and the MHS-MXP algorithm. We start from the existing experimental implementation of the abduction solver, which allows the use of both algorithms. To perform consistency checking and model extraction, these algorithms need to call an external DL reasoner, which is used as a black box. However, model extraction did not work properly in the original implementation. Our goal is to solve this problem while trying to maintain the black box approach. The problem is not trivial, as model extraction does not belong to the standard tasks of DL reasoners. In our work, we explored possible solutions and finally chose the `OWLKnowledgeExplorerReasoner` interface. In addition, we improved the solver implementation and expanded the types of explanations, defined relevance for multiple observations, modified the outputs and fixed a few serious bugs. In the end, an empirical evaluation, consisting of two experiments, was carried out. In the first experiment, we compared the MHS and MHS-MXP algorithms and showed the type of inputs on which MHS-MXP is significantly better, but also the type of inputs that are problematic for it. In the second experiment, we tested the MHS-MXP algorithm on a more varied sample of ontologies.

Keywords: abduction, model extraction, MHS algorithm, MHS-MXP algorithm

Abstrakt

Abdukcia je typ inferencie, ktorého úlohou je vysvetliť nejaké pozorovanie na základe znalostí, ktoré máme. Na riešenie abdukčného problému bolo navrhnutých viacero algoritmov. My sa zaoberáme štandardným MHS algoritmom a MHS-MXP algoritmom. Vychádzame z existujúcej experimentálnej implementácie abdukčného solveru, ktorý umožňuje použitie oboch algoritmov. Tieto algoritmy potrebujú pre zistenie konzistencie a extrahovanie modelu volať externý DL reasoner, ktorý je využívaný ako čierna skrinka. Extrakcia modelov však v pôvodnej implementácii nefungovala správne. Naším cieľom je vyriešiť tento problém so snahou zachovať prístup k reasoneru ako k čiernej skrinke. Nejedná sa o triviálny problém, pretože extrakcia modelu nepatrí k štandardných úlohám DL reasonerov. V práci sme preskúmali možné riešenia a nakoniec vybrali `OWLKnowledgeExplorerReasoner` rozhranie. Okrem toho sme vylepšili implementáciu solvera a rozšírili typy vysvetlení, definovali relevanciu pre mnohonásobné pozorovanie, upravili výstupy a opravili pár závažnejších chýb. Na konci bola vykonaná empirická evalvácia pozostávajúca z dvoch experimentov. V prvom experimente sme porovnali MHS a MHS-MXP algortimus a ukázali sme typ vstupov, na ktorých je MHS-MXP výrazne lepší, ale aj typ vstupov, ktoré sú pre neho problematické. V druhom experimente sme MHS-MXP algortimus vyskúšali na pestrejšej vzorke ontológií.

Kľúčové slová: abdukcia, extrakcia modelov, MHS algoritmus, MHS-MXP algoritmus

Contents

Introduction **1**

I State of the art **3**

1 Description Logics **4**

1.1 Syntax of $\mathcal{ALCHO}$ 5

1.2 Semantics of $\mathcal{ALCHO}$ 10

1.3 Basic decision problems 15

1.4 Tableau algorithm for $\mathcal{ALCHO}$ 18

1.4.1 CTree 19

1.4.2 CTree initialization and expansion 20

1.5 Ontologies 26

2 Abduction **28**

2.1 Abduction problem 29

2.1.1 Observation types 30

2.1.2 Abducibles 31

2.2 Methods for searching explanations 32

2.2.1 MHS algorithm 33

2.2.2 Hybrid algorithm 39

3 Original implementation **49**

3.1 Running the solver 50

3.2 Inputs 50
3.3 Outputs 52
3.4 Implementation structure 55
3.5 Model extraction 55

II Our contribution 58

4 Model extraction 59
4.1 OWLKnowledgeExplorerReasoner 59
4.2 DIG Interface 60
4.3 JFact Reasoner 61

5 Implementation 64
5.1 Model extraction 64
5.1.1 Fixing JFact implementation of the interface 65
5.1.2 Obtaining the concept assertions of a model 66
5.1.3 Obtaining the role assertions of a model 68
5.2 Relevance for multiple observations 68
5.3 Log files adjustment 71
5.3.1 Location of logging files 71
5.3.2 New types of log files 72
5.4 Bug fixes and improvements 72
5.4.1 Interpretation of the consistency check result 72
5.4.2 Consistent explanations check 73

6 Evaluation 75
6.1 Experiment 1 76
6.1.1 Methodology 76
6.1.2 Results 79
6.2 Experiment 2 83
6.2.1 Methodology 83
6.2.2 Results 87

Conclusion **89**

Bibliography **91**

Introduction

Abduction [7, 16] belongs to less standard types of reasoning. It consists in finding an explanation for an observation based on background knowledge that we have. It can be also perceived as backward reasoning, where we try to get from an observation to its cause. A basic example would be a medical diagnosis: our observation is the patient's symptoms, and then based on some medical background knowledge, we want to get a diagnosis that causes them. It is important to note that abduction is hypothetical reasoning, thus its conclusions do not necessarily have to be true. In addition to medicine, abduction could also be used in many other areas, such as in criminology or to reveal the reason for a system malfunction.

In knowledge representation, ontologies [8] are often used. In this context, we can imagine an ontology as a structure that serves to represent the knowledge of a certain application domain. We will focus on ontologies that are based on formalisms called description logics [21, 1] because they belong to the most popular ones [3]. Description logics (DLs) are a family of formal languages which are (mostly) decidable fragments of first-order logic.

Several algorithms have been proposed to solve the abduction problem for DLs. We will deal with two of them: the well-known standard MHS algorithm [20, 17] and the MHS-MXP algorithm [10]. These algorithms are complete, which means that they can find all explanations of the abduction problem. For both algorithms, it is necessary to perform a consistency check and model extraction many times. These reasoning tasks are ensured by an external DL reasoner which is used as a black box.

We have started from an existing implementation of the abduction solver [4], in which either MHS or MHS-MXP can be used to find explanations for the abduction problem. The critical problem of the solver was model extraction, which was not performed correctly. The following goal was to perform an evaluation, which consists of two parts: (1) repeating the previous evaluation [4], but this time with the correct model extraction and the inclusion of another group of inputs and (2) evaluating MHS-MXP on real-world ontologies for the first time.

In our work, we explored several possible ways to solve the problem of model extraction. In the end, we chose the `OWLKnowledgeExplorerReasoner` interface[1], however, we needed to alter its implementation to make it work. This result is very significant because before it was not possible to solve this problem while maintaining the black box approach. Next, we improved the implementation by expanding the possible explanations, adjusting solver outputs and fixing a couple of hidden bugs. Finally, we performed an empirical evaluation with two experiments: In Experiment 1, we repeated the previous evaluation and compared MHS and MHS-MXP. We found that on one group of inputs, MHS-MXP significantly outperformed MHS, but the other group was problematic for MHS-MXP. In Experiment 2, we evaluated MHS-MXP on a group of real-world ontologies that were selected from ORE 2015 Reasoner Competition Corpus[2].

In Chapter 1, we will explain description logics and briefly talk about ontologies. In Chapter 2, we will introduce abduction, the abduction problem and both the MHS and the MHS-MXP algorithm. In Chapter 3, we will look at the original implementation of our solver. In Chapter 4, we will present possible solutions for model extraction. In Chapter 5, we will focus on our contribution to the solver implementation. Lastly, in Chapter 6, we will look at the conducted empirical evaluation.

[1] `https://owlcs.github.io/owlapi/apidocs_4/org/semanticweb/owlapi/reasoner/knowledgeexploration/OWLKnowledgeExplorerReasoner.html`

[2] `https://zenodo.org/record/18578#.Y3tygXbMJPb`

Part I

State of the art

Chapter 1

Description Logics

Description logics (DLs) [21, 2, 1] are a family of languages used for knowledge representation. They formally describe an application domain in a structured, clear and understandable way. The word *description* from the name is derived from the fact that an application domain is represented by concept *descriptors*, which are the expressions created from atomic concepts (unary predicates) and atomic roles (binary predicates) connected with the constructors of the specific DL language. The word *logic* from the name comes from the logic-based semantics of DLs that is pretty similar to first-order logic (FOL) semantics.

DLs have their predecessors, early knowledge representations, such as semantic networks and frames, which were very intuitive and comprehensive so they were readable and easily understood. On the other hand, they did not have a precise meaning and their interpretation could differ from person to person. This caused trouble while reasoning. Therefore, DLs were developed with the aim of preserving their intuitive representation and adding formal semantics to overcome the ambiguity of interpretation.

Unlike FOL, most DL languages are decidable (Definition 1.0.1). Research in the field of DL is mainly interested in decidable DL fragments, so decidability becomes a necessary condition for DL languages.

Definition 1.0.1 (Decidability [21]) *A class of problems is called decidable if there is a generic algorithm that can take any problem instance as an input and provide a yes-or-no answer after a finite time.*

In the context of logics, the common generic problem is entailment. If any of the logic problems is decidable, sometimes the logic itself is called decidable. As already mentioned, there are different DL languages. They differ in constructors that they use. More constructors typically mean more expressivity. In this chapter, we will focus on a specific DL language called $\mathcal{ALCHO}$, which is the most suitable for understanding our work. We will go through its syntax, semantics and other important notions.

1.1 Syntax of $\mathcal{ALCHO}$

DL is composed of three basic types of entities: individual names, concept names and role names. **Individual names** represent concrete objects from an application domain. For example, a concrete person tim, a pet fluffy or an object stool527. **Concept names** contain names that represent types or categories of objects that usually have similar properties. They also can be viewed as classes of objects. For instance, we can have concept name Person, Professor, Pet or Furniture. **Role names** contain names that represent binary relations that can occur between any two objects of a domain. For example, hasPet or teaches. We usually refer to these three sets of entities as **Vocabulary** (Definition 1.1.1).

Definition 1.1.1 (Vocabulary) *A DL vocabulary consists of three countable mutually disjoint sets of symbols:*

- *set of individuals* $N_I = \{a, b, ...\}$
- *set of atomic concepts* $N_C = \{A, B, ...\}$
- *set of roles* $N_R = \{R, S, ...\}$

According to a convention, concept names are capitalized while individual and role names are written in lowercase. Camel case is used for names that were created from multi-word notions.

Example 1.1.1 (Vocabulary) *Let us have a domain of pets and people. Vocabulary for this domain can be:*

- $N_I = \{\mathsf{tim}, \mathsf{eva}, \mathsf{erik}, \mathsf{fluffy}, \mathsf{pluto}, \mathsf{falco}\}$
- $N_C = \{\mathsf{Person}, \mathsf{Owner}, \mathsf{Good}, \mathsf{Happy}, \mathsf{Pet}, \mathsf{Dog}, \mathsf{Cat}\}$
- $N_R = \{\mathsf{likes}, \mathsf{owns}\}$

In $\mathcal{ALCHO}$ DL it is possible to define the concept by enumerating the individuals that should belong to it. These types of concepts are referred to as **nominals** (Definition 1.1.2).

Definition 1.1.2 (Nominals) *Nominals (nominal concepts) are concept expressions of the form:*

$$\{a_1, a_2, ..., a_n\}$$

where $\{a_1, a_2, ..., a_n\} \subseteq N_I$.

We can use nominals when we want to refer to some concrete individuals. That way we do not have to create an artificial concept name for these individuals.

Sometimes it is necessary to express more complex descriptions. We may, for instance, want to express happy people from our example domain without the need to create another concept name which would correspond to it. For these purposes, we have complex concepts. **Complex concepts** (Definition 1.1.3), also called non-atomic concepts or concepts, are recursively constructed from *atomic concepts* and *constructors*. Different DL languages use different sets of constructors to create complex concepts. Constructors used in $\mathcal{ALCHO}$ DL are: complement ($\neg$), intersection ($\sqcap$), union ($\sqcup$), existential restriction ($\exists$), value restriction ($\forall$).

The meaning of these constructors is very similar to the operations of the same name from set theory.

Definition 1.1.3 (Complex concepts) *Concepts are recursively constructed as the smallest set of expressions of the forms:*

$$C, D ::= A \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C \mid \{a\}$$

where $a \in N_I, A \in N_C, R \in N_R$*, and* C*,* D *are concepts.*

Example 1.1.2 (Complex concepts and nominals) *We use the vocabulary from Example 1.1.1.*

$\neg$**Person** *This complex concept refers to all objects that are not people.*

Happy $\sqcap$ **Person** *This complex concept refers to all objects that are happy and people at the same time. So, simply put, it refers to happy people.*

Person $\sqcup$ **Pet** *This complex concept refers to all objects that are people or pets.*

$\exists$**hasPet**.**Dog** *This complex concept refers to all objects that are in relation* hasPet *with an object that is a dog. So, in natural language, we can say that it corresponds to all people that have a dog as a pet.*

$\forall$**likes**.{**falco**} *This complex concept refers to all objects for which holds that if they like anything it has to be the individual* falco*. So, it refers to all objects that like only* falco *or nothing at all.*

$\exists$**hasPet**.**Dog** $\sqcap$ $\forall$**likes**.{**falco**, **fluffy**} *We can also create a concept from complex concepts. This concept refers to all objects that have a dog as a pet and, at the same time, any object they like is either* falco *or* fluffy*.*

Another simplification, syntactic sugar, is the **top concept** and **bottom concept** (Definition 1.1.4). In some cases, we may want to refer to all objects from a domain (top concept), or, on the contrary, we want to express that

something does not apply to any object, that is, to somehow express an empty set (bottom concept).

Definition 1.1.4 (Top and bottom concepts) *The top ($\top$) and bottom ($\bot$) concepts are defined as syntactic shorthands:*

- $\top$ *is a placeholder for* $A \sqcup \neg A$
- $\bot$ *is a placeholder for* $A \sqcap \neg A$

where A is any atomic concept.

Now that we are familiar with the basic entity types and concepts, it is time to look at how we formally represent domain knowledge. This domain knowledge is captured in a **knowledge base** (Definition 1.1.7).

We distinguish between two basic types of knowledge: *intensional knowledge* and *extensional knowledge*. Intensional knowledge is general knowledge or terminology of a domain that describes concepts and roles, and relationships between them. For example, we can have a relationship between the concepts Owner and Person: all owners are people, and that would be part of intensional knowledge. Extensional knowledge is knowledge about individuals and is often referred to as empirical knowledge or facts. For instance, we can say that individual fluffy is a pet.

Because of this knowledge division, the knowledge base is also divided into two parts: **TBox** (Definition 1.1.5) and **ABox** (Definition 1.1.6). TBox contains intensional knowledge while ABox contains extensional knowledge.

Definition 1.1.5 (TBox) *A TBox $\mathcal{T}$ is a finite set of GCI and RIA axioms ϕ of the form:*

- $\phi ::= C \sqsubseteq D$
- $\phi ::= R \sqsubseteq S$

where C, D are any concepts and $R, S \in N_R$.

The term GCI stands for general concept inclusion and term RIA for role inclusion axiom. They are subsumption axioms.

Definition 1.1.6 (ABox) *An ABox $\mathcal{A}$ is a finite set of assertion axioms (assertions) ϕ of the form:*

- $\phi ::= a\colon C$ *(concept assertion)*
- $\phi ::= a, b\colon R$ *(role assertion)*

where $a, b \in N_I, R \in N_R$, and C is any concept.

Let us also define assertion $a, b\colon \neg R$ as a shortcut for $a\colon \forall R.\neg\{b\}$, where $a, b \in N_I$ and $R \in N_R$ (Lemma 1.1.1). We will refer to this expression as a *negated role assertion*. The equality of these expressions will be shown in Proof 1.2.1.

Lemma 1.1.1 *Expressions $a\colon \forall R.\neg\{b\}$ and $a, b\colon \neg R$ are equivalent.*

We can come across an alternative notation of assertions, that is often used. Concept assertion might be denoted as $C(a)$ where $a \in N_I$ and C is any concept. Role assertion might be denoted as $R(a, b)$ where $a, b \in N_I$ and $R \in N_R$. These alternative notations have the exact same meaning as those from the definition and might be used interchangeably.

Definition 1.1.7 (Knowledge base) *A DL knowledge base (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is a pair consisting of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$.*

In the following chapters, we can encounter a knowledge base $\mathcal{K}$ expressed as a set, then it is a simplification, where $\mathcal{K} = \mathcal{T} \cup \mathcal{A}$.

Example 1.1.3 (Knowledge base) *We will use vocabulary from Example 1.1.1 and create a concise KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ about people and pets.*

$$\begin{aligned}\mathcal{T} = \{&\mathsf{Owner} \sqsubseteq \mathsf{Person},\\ &\mathsf{Owner} \sqsubseteq \exists \mathsf{owns}.\mathsf{Pet},\end{aligned}$$

$$\begin{aligned} \mathsf{Dog} \sqcup \mathsf{Cat} &\sqsubseteq \mathsf{Pet}, \\ \mathsf{owns} &\sqsubseteq \mathsf{likes}, \\ \exists \mathsf{owns}.(\mathsf{Good} \sqcap \mathsf{Pet}) &\sqsubseteq \mathsf{Happy}\} \end{aligned}$$

$$\begin{aligned} \mathcal{A} = \{&\mathsf{eva} : \mathsf{Owner}, \\ &\mathsf{fluffy} : \mathsf{Dog}, \\ &\mathsf{fluffy} : \mathsf{Good}, \\ &\mathsf{tim}, \mathsf{fluffy} : \mathsf{owns}\} \end{aligned}$$

In some materials [21], we can also see a different division of the knowledge base: *Abox*, *TBox* and *RBox*. In such cases, the axioms that describe the roles and their relationships are contained in RBox. In $\mathcal{ALCHO}$ DL, RBox would contain RIA axioms. The TBox would then contain only descriptions of the concepts and their relationships, so GCI axioms. Thus, intensional knowledge would be divided into TBox and RBox.

If we were to apply this different division to our knowledge base from Example 1.1.3, the only change would be to move the axiom $\mathsf{owns} \sqsubseteq \mathsf{likes}$ to RBox.

1.2 Semantics of $\mathcal{ALCHO}$

Now that we got familiar with the syntax, we will look at the $\mathcal{ALCHO}$ DL semantics. DLs semantics belongs to model-theoretic semantics. In this type of semantics, models are used to interpret symbols of the language. A model is a mathematical structure that represents some possible "state of the world". Usually, it is not the state of the whole world but only some small part, an application domain. A model maps symbols from the vocabulary to the model elements. It consists of a *domain* which contains elements and an *interpretation function* which ensures the mapping. We can imagine model elements to be some concrete objects of our world. They can be represented

in different ways, e.g. numbers, words or pictures. The number of these elements can be infinite. The interpretation function then maps the symbols to elements from the model domain. Every symbol from the vocabulary has to have its interpretation. However, there is no need to use all elements from the domain to interpret something.

In DLs, these model structures are called **interpretations** (Definition 1.2.1) and they are used to describe the meaning of DL entities.

As we mentioned in the chapter introduction, the semantics of DLs are very similar to FOL semantics. FOL also has model-theoretic semantics. Instead of the term *interpretation*, FOL uses the term *structure* to denote model structure.

Definition 1.2.1 (Interpretation) *An interpretation of a given vocabulary is a pair* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *which contains:*

- *a non-empty* ***domain*** $\Delta^{\mathcal{I}}$
- *an* ***interpretation function*** $\cdot^{\mathcal{I}}$*, such that:*
  - $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ *for all* $a \in N_I$
  - $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ *for all* $A \in N_C$
  - $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ *for all* $R \in N_R$
- *for any concepts C, D, role R and individuals* $a_1, ..., a_n \in N_I$ *the* ***interpretation of complex concepts*** *is recursively defined as follows:*
  - $\neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
  - $C \sqcap D^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
  - $C \sqcup D^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
  - $\exists R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}} : (x, y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
  - $\forall R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x, y) \in R^{\mathcal{I}} \implies y \in C^{\mathcal{I}}\}$
  - $\{a_1, ..., a_n\}^{\mathcal{I}} = \{a_1^{\mathcal{I}}, ..., a_n^{\mathcal{I}}\}$

Example 1.2.1 (Interpretation) *In the example, we will show an interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *of the vocabulary from Example 1.1.1.*

$$\Delta^{\mathcal{I}} = \{\ 🧒, 🧔, 👧, 👩, 👴, 🧓, 👵, 🐶, 🐕, 🐩, 🦮, 🐱, 🐈, 🐈‍⬛\ \},$$

$$\begin{aligned}
\mathsf{tim}^{\mathcal{I}} &= 🧒, & \mathsf{Person}^{\mathcal{I}} &= \{👧\}, & \mathsf{owns}^{\mathcal{I}} &= \{(🧒,🐶),(👧,🐈‍⬛)\},\\
\mathsf{eva}^{\mathcal{I}} &= 👧, & \mathsf{Owner}^{\mathcal{I}} &= \{👧,🧒\}, & \mathsf{likes}^{\mathcal{I}} &= \{(👧,🐈‍⬛)\}\\
\mathsf{erik}^{\mathcal{I}} &= 👴, & \mathsf{Happy}^{\mathcal{I}} &= \{🧒\},\\
\mathsf{fluffy}^{\mathcal{I}} &= 🐶, & \mathsf{Good}^{\mathcal{I}} &= \{🐶\},\\
\mathsf{pluto}^{\mathcal{I}} &= 🐈‍⬛, & \mathsf{Pet}^{\mathcal{I}} &= \{🐈‍⬛\},\\
\mathsf{falco}^{\mathcal{I}} &= 🦮, & \mathsf{Dog}^{\mathcal{I}} &= \{🐶\},\\
& & \mathsf{Cat}^{\mathcal{I}} &= \{\},
\end{aligned}$$

Interpretation $\mathcal{I}$ *is only one possible example. There are infinite possible interpretations of a given vocabulary, in which also different domains can be used.*

In addition to symbols, we must also look at axioms in the context of interpretations. Axioms are some kind of statements usually given in the form of a knowledge base. Since the interpretation is some concrete state of the world, it is interesting for us to know whether these statements are valid in the given interpretation. So, we are interested in whether the axiom **is satisfied** in the given interpretation (Definition 1.2.2).

Definition 1.2.2 (Satisfaction) *Given an axiom* ϕ*, an interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *satisfies* ϕ $(\mathcal{I} \models \phi)$ *depending on its type:*

- $\mathbf{C} \sqsubseteq \mathbf{D} : \mathcal{I} \models C \sqsubseteq D$ *iff* $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
- $\mathbf{R} \sqsubseteq \mathbf{S} : \mathcal{I} \models R \sqsubseteq S$ *iff* $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
- $\mathbf{a}\colon \mathbf{C} : \mathcal{I} \models a\colon C$ *iff* $a^{\mathcal{I}} \in C^{\mathcal{I}}$
- $\mathbf{a}, \mathbf{b}\colon \mathbf{R} : \mathcal{I} \models a, b\colon R$ *iff* $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$

where C, D are any concepts, R, $S \in N_R$ and $a,b \in N_I$.

Proof 1.2.1 (Derivation of negated role assertion) *Let us show, using semantics, that the expressions $a\colon \forall R.\neg\{b\}$ and $a, b\colon \neg R$ are equivalent.*

Let us have an interpretation $\mathcal{I}$ such that $\mathcal{I} \models a\colon \forall R.\neg\{b\}$.

Then $\mathcal{I} \models a\colon \forall R.\neg\{b\}$ iff $a^{\mathcal{I}} \in \forall R.\neg\{b\}^{\mathcal{I}}$

$$\begin{aligned}\forall R.\neg\{b\}^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \implies y \in \neg\{b\}^{\mathcal{I}}\} \\ &= \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \implies y \in \Delta^{\mathcal{I}} \setminus \{b^{\mathcal{I}}\}\} \\ &= \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \implies y \neq b^{\mathcal{I}}\}\end{aligned}$$

So $a^{\mathcal{I}} \in \{x \in \Delta^{\mathcal{I}} \mid \forall y \in \Delta^{\mathcal{I}} : (x,y) \in R^{\mathcal{I}} \implies y \neq b^{\mathcal{I}}\}$. This means that if a is assigned in a role R with any individual, it cannot be b. Therefore a, b must not belong to R and it must belong to its complement $\neg R$. From that $\mathcal{I} \models a, b\colon \neg R$.

Since we have defined the satisfaction of axioms, now we can determine whether a given interpretation satisfies a knowledge base. An interpretation that satisfies a knowledge base is called a **model** of a given knowledge base (not to be confused with the term *model as a mathematical structure* mentioned at the beginning of this section). We can imagine knowledge base models as the states of the world in which the statements from the knowledge base hold.

Definition 1.2.3 (Model) *An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model of a DL KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ (denoted $\mathcal{I} \models \mathcal{K}$) iff $\mathcal{I}$ satisfies every axiom in $\mathcal{T}$ and $\mathcal{A}$.*

Example 1.2.2 (Not a model) *Let us take interpretation $\mathcal{I}$ from Example 1.2.1 and show that it is <u>not</u> a model of KB $\mathcal{K}$ from Example 1.1.3. For $\mathcal{I}$ to be the model of the $\mathcal{K}$, it has to satisfy all axioms from $\mathcal{K}$. This is not the case. For example axiom* $\mathbf{Dog} \sqcup \mathbf{Cat} \sqsubseteq \mathbf{Pet}$ *is <u>**not satisfied**</u>:*

$$\mathcal{I} \models \mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet} \text{ iff } \mathsf{Dog} \sqcup \mathsf{Cat}^{\mathcal{I}} \subseteq \mathsf{Pet}^{\mathcal{I}}$$

First, we formulate interpretations of the concepts $\mathsf{Dog} \sqcup \mathsf{Cat}$ *and* Pet.

$$\mathsf{Dog} \sqcup \mathsf{Cat}^{\mathcal{I}} = \mathsf{Dog}^{\mathcal{I}} \cup \mathsf{Cat}^{\mathcal{I}} = \{🐶\},$$
$$\mathsf{Pet}^{\mathcal{I}} = \{🐈‍⬛\}$$

We see that $\{🐶\} \not\subseteq \{🐈‍⬛\}$, *so* $\mathcal{I} \not\models \mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$ *and then* $\mathcal{I} \not\models \mathcal{K}$.

Note that some other axioms were also not satisfied, for example $\mathsf{owns} \sqsubseteq \mathsf{likes}$. *However, it is sufficient to show that it does not hold for at least one to prove that interpretation is not a model of* $\mathcal{K}$.

Example 1.2.3 (Model) *In this example we will show interpretation* $\mathcal{I}_2 = (\Delta^{\mathcal{I}_2}, \cdot^{\mathcal{I}_2})$ *that it is a model of KB* $\mathcal{K}$ *from Example 1.1.3.*

$$\Delta^{\mathcal{I}_2} = \{\ 🧒, 🧔, 👧, 👩, 👴, 👵, 👴, 🐶, 🐕, 🐩, 🦮, 🐱, 🐈, 🐈‍⬛\ \},$$

$$\mathsf{tim}^{\mathcal{I}_2} = 🧒, \quad \mathsf{Person}^{\mathcal{I}_2} = \{👧,🧒\}, \quad \mathsf{owns}^{\mathcal{I}_2} = \{(🧒,🐶),(👧,🐈‍⬛)\},$$
$$\mathsf{eva}^{\mathcal{I}_2} = 👧, \quad \mathsf{Owner}^{\mathcal{I}_2} = \{👧,🧒\}, \quad \mathsf{likes}^{\mathcal{I}_2} = \{(🧒,🐶),(👧,🐈‍⬛)\},$$
$$\mathsf{erik}^{\mathcal{I}_2} = 👴, \quad \mathsf{Happy}^{\mathcal{I}_2} = \{🧒\},$$
$$\mathsf{fluffy}^{\mathcal{I}_2} = 🐶, \quad \mathsf{Good}^{\mathcal{I}_2} = \{🐶\},$$
$$\mathsf{pluto}^{\mathcal{I}_2} = 🐈‍⬛, \quad \mathsf{Pet}^{\mathcal{I}_2} = \{🐶,🐈‍⬛\},$$
$$\mathsf{falco}^{\mathcal{I}_2} = 🦮, \quad \mathsf{Dog}^{\mathcal{I}_2} = \{🐶\},$$
$$\mathsf{Cat}^{\mathcal{I}_2} = \{\},$$

In order to prove that $\mathcal{I}_2$ *is indeed a model of* $\mathcal{K}$, *we would have to prove that it satisfies* every *axiom from* $\mathcal{K}$. *To save space, we will not provide the whole proof, only demonstrate the principle on a few axioms.*

eva: **Owner** :
$\mathcal{I}_2 \models \mathsf{eva} : \mathsf{Owner}$ *iff* $\mathsf{eva}^{\mathcal{I}_2} \in \mathsf{Owner}^{\mathcal{I}_2}$
$👧 \in \{👧,🧒\}$, *so* $\mathcal{I}_2 \models \mathsf{eva} : \mathsf{Owner}$

tim, fluffy : owns :
$\mathcal{I}_2 \models \mathsf{tim}, \mathsf{fluffy} : \mathsf{owns}$ *iff* $(\mathsf{tim}^{\mathcal{I}_2}, \mathsf{fluffy}^{\mathcal{I}_2}) \in \mathsf{owns}^{\mathcal{I}_2}$
$(🧒,🐶) \in \{(🧒,🐶),(👧,🐈)\}$, *so* $\mathcal{I}_2 \models \mathsf{tim}, \mathsf{fluffy} : \mathsf{owns}$

Owner $\sqsubseteq \exists$**owns.Pet** :
$\mathcal{I}_2 \models \mathsf{Owner} \sqsubseteq \exists\mathsf{owns.Pet}$ *iff* $\mathsf{Owner}^{\mathcal{I}_2} \subseteq \exists\mathsf{owns.Pet}^{\mathcal{I}_2}$
$\exists\mathsf{owns.Pet}^{\mathcal{I}_2} = \{x \in \Delta^{\mathcal{I}_2} \mid \exists y \in \Delta^{\mathcal{I}_2} : (x,y) \in \mathsf{owns}^{\mathcal{I}_2} \wedge y \in \mathsf{Pet}^{\mathcal{I}_2}\}$
$= \{x \in \Delta^{\mathcal{I}_2} \mid \exists y \in \Delta^{\mathcal{I}_2} : (x,y) \in \{(🧒,🐶),(👧,🐈)\} \wedge y \in \{🐶,🐈\}\}$
$= \{👧,🧒\}$
$\{👧,🧒\} \subseteq \{👧,🧒\}$, *so* $\mathcal{I}_2 \models \mathsf{Owner} \sqsubseteq \exists\mathsf{owns.Pet}$

Dog $\sqcup$ **Cat** $\sqsubseteq$ **Pet** :
$\mathcal{I}_2 \models \mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$ *iff* $\mathsf{Dog} \sqcup \mathsf{Cat}^{\mathcal{I}_2} \subseteq \mathsf{Pet}^{\mathcal{I}_2}$
$\mathsf{Dog} \sqcup \mathsf{Cat}^{\mathcal{I}_2} = \mathsf{Dog}^{\mathcal{I}_2} \cup \mathsf{Cat}^{\mathcal{I}_2} = \{🐶\}$
$\{🐶\} \subseteq \{🐶,🐈\}$, *so* $\mathcal{I}_2 \models \mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$

1.3 Basic decision problems

In the previous sections, we showed how to represent knowledge using language $\mathcal{ALCHO}$ DL. However, that is not the only thing we can do. Once we capture knowledge in a logical representation, a DL knowledge base, we can further reason with this knowledge.

The basic reasoning task is drawing *consequences* (implicit knowledge) from explicit knowledge that is captured in a knowledge base. This means that when we have a statement in the form of an axiom and we want to know whether this statement follows from a knowledge base. This decision problem is called **entailment** (Definition1.3.1).

Definition 1.3.1 (Entailment (Logical consequence)) *An axiom ϕ is entailed by a KB $\mathcal{K}$ (denoted $\mathcal{K} \models \phi$) iff for every $\mathcal{I}$, such that $\mathcal{I} \models \mathcal{K}$, holds $\mathcal{I} \models \phi$.*

In our case, it is enough to limit ourselves to problems related to ABox. Therefore, we will be interested in the entailment of assertion axioms. This

decision problem is also referred to as **instance checking**. There are two types of assertion axioms, so we also have two types of instance checking: *concept instance checking* (Definition 1.3.2) and *role instance checking* (Definition 1.3.3).

Definition 1.3.2 (Concept instance checking) *An individual a is an instance of a concept C w.r.t. a DL KB $\mathcal{K}$ (denoted $\mathcal{K} \models a\colon C$) iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ in all models $\mathcal{I}$ of $\mathcal{K}$.*

Definition 1.3.3 (Role instance checking) *A pair of individuals (a, b) is an instance of a role R w.r.t. a DL KB $\mathcal{K}$ (denoted $\mathcal{K} \models (a, b)\colon R$) iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ in all models $\mathcal{I}$ of $\mathcal{K}$.*

Another important reasoning task is to decide if a knowledge base is consistent (Definition 1.3.4).

Definition 1.3.4 (ABox consistency) *A DL KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent (also, $\mathcal{A}$ is consistent w.r.t. $\mathcal{T}$) iff it has at least one model.*

There may, indeed, be knowledge bases in which there is some kind of conflict and then it is not possible to construct an interpretation that would satisfy them. Let us show it in Example 1.3.1.

Example 1.3.1 (Inconsistent knowledge base) *We will use vocabulary from Example 1.1.1. Let $\mathcal{K}_2 = (\mathcal{T}_2, \mathcal{A}_2)$ be our DL knowledge base, such that:*

$$\mathcal{T}_2 = \{\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}, \exists \mathsf{owns}.(\mathsf{Good} \sqcap \mathsf{Pet}) \sqsubseteq \mathsf{Happy}\}$$
$$\mathcal{A}_2 = \{\mathsf{fluffy}\colon \mathsf{Dog}, \mathsf{fluffy}\colon \mathsf{Good}, \mathsf{tim}\colon \neg\mathsf{Happy}, \mathsf{tim}, \mathsf{fluffy}\colon \mathsf{owns}\}$$

We can try to create a model of $\mathcal{K}_2$, but eventually, we will run into a conflict.

Individual fluffy *belongs to concept* Dog*, so from axiom* $\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$ *it belongs to concept* Pet*. So individual* fluffy *belongs to concepts* Good *and* Pet*.*

Individual tim *is in relation* owns *with* fluffy*, therefore* tim *belongs to concept* $\exists\mathsf{owns}.(\mathsf{Good} \sqcap \mathsf{Pet})$*. Then, according to axiom* $\exists\mathsf{owns}.(\mathsf{Good} \sqcap \mathsf{Pet}) \sqsubseteq \mathsf{Happy}$*,* tim *must belong to concept* Happy*.*

In $\mathcal{K}_2$*, however,* tim *belongs to concept* $\neg\mathsf{Happy}$*. So, there is a conflict,* tim *cannot belong to* Happy *and* $\neg\mathsf{Happy}$ *at the same time. We are not able to create any model, so* $\mathcal{K}_2$ *is inconsistent.*

Some decision problems can be simplified and reformulated into another decision problem that is easier to solve. We can, for example, reduce the problem of instance checking to a consistency check (Lemmata 1.3.1, 1.3.2).

Lemma 1.3.1 (Reduction of concept instance checking) *Given a DL KB* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$*, an individual* a *and a concept* C*:*

$$\mathcal{K} \models a\colon C \text{ iff } \mathcal{K}' = (\mathcal{T}, \mathcal{A} \cup \{a\colon \neg C\}) \text{ is inconsistent.}$$

Lemma 1.3.2 (Reduction of role instance checking) *Given a DL KB* $\mathcal{K} = (\mathcal{T}, \mathcal{A})$*, individuals* a, b *and a role* R*:*

$$\mathcal{K} \models (a, b)\colon R \text{ iff } \mathcal{K}' = (\mathcal{T}, \mathcal{A} \cup \{(a, b)\colon \neg R\}) \text{ is inconsistent.}$$

These lemmata come from an idea based on the definition of entailment and consistency. An entailed axiom must be satisfied in all models of knowledge base $\mathcal{K}$. If we add its negation to $\mathcal{K}$, then we should come to a conflict, because it is not possible to satisfy an axiom and its negation at the same time.

Example 1.3.2 (Concept instance checking) *We will use vocabulary from Example 1.1.1. Let* $\mathcal{K}_3 = (\mathcal{T}_3, \mathcal{A}_3)$ *be our DL knowledge base, such that:*

$$\mathcal{T}_3 = \{\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}\}$$
$$\mathcal{A}_3 = \{\mathsf{fluffy}\colon \mathsf{Dog}\}$$

Let us show that axiom $\mathbf{fluffy}\colon \mathbf{Pet}$ ***is entailed by*** $\mathcal{K}_3$*. We will proceed according to lemma 1.3.1. Firstly, we construct a new knowledge base* $\mathcal{K}'_3 =$

$(\mathcal{T}_3, \mathcal{A}_3 \cup \{\mathsf{fluffy}\colon \neg\mathsf{Pet}\})$. *Secondly, we show that* $\mathcal{K}'_3$ *is inconsistent. We try to construct a model of* $\mathcal{K}'_3$*, but eventually, we will run into a conflict.*

Individual fluffy *belongs to concept* Dog*, so from axiom* $\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$ *he belongs to concept* Pet*. In* $\mathcal{K}'_3$*, however,* fluffy *belongs to concept* $\neg$Pet*. So, there is a conflict because* fluffy *cannot belong to both* Pet *and* $\neg$Pet*. We are not able to create any model, so* $\mathcal{K}'_3$ *is inconsistent and from that* $\mathcal{K}_3 \models \mathsf{fluffy}\colon \mathsf{Pet}$*.*

Finally, it is important to note that if a knowledge base is inconsistent, then it entails any axiom.

1.4 Tableau algorithm for $\mathcal{ALCHO}$

In the previous section, we got acquainted with two common decision problems and showed examples of how to solve them intuitively. In this section, we will introduce a reasoning algorithm that can be used to check the consistency of a KB (and therefore also instance checking, according to Lemmata 1.3.1 and 1.3.2).

We will describe a commonly used reasoning algorithm called *Tableau algorithm* [19, 21, 4, 15, 1]. It is a model-theoretic reasoning method, thus in order to prove the consistency of a given KB, it tries to construct a model that satisfies all the KB axioms. If it succeeds, the KB is consistent, otherwise, the KB is inconsistent.

The advantage of the algorithm is that it can construct a finite model representation and thus verify consistency even if a given KB has only infinite models. We can read a partial finite model from the resulting structure and also obtain information on how it would be expanded to create the real infinite model.

1.4.1 CTree

The structure created by the tableau algorithm is called a **completion tree** (Definition 1.4.1) or a **completion graph** because in general, the structure is a graph, not necessarily a tree. A completion tree consists of *vertices* and *edges*. Both vertices and edges are marked with a *label*.

Definition 1.4.1 (Completion tree) *A completion tree (CTree) is a triple* $T = (V, E, \mathcal{L})$*, where* V *is a set of vertices,* E *is a set of edges and* $\mathcal{L}$ *is a labelling function s.t.*

- $\mathcal{L}(x)$ *is a set of concepts for all* $x \in V$
- $\mathcal{L}((x, y))$ *is a set of roles for all* $(x, y) \in E$

We can also look at a CTree T as a representation of an interpretation $\mathcal{I}$, where the vertices represent the elements from the domain $\Delta^{\mathcal{I}}$ and edges represent relationships between them. Then the label of the vertex $x \in V$ indicates to which concepts the corresponding element $x \in \Delta^{\mathcal{I}}$ belongs. The label of the edge (x, y) contains the names of the relationships (the role names) between corresponding elements $x, y \in \Delta^{\mathcal{I}}$.

For tableau algorithm to work, it is required that vertex labels in a CTree must contain **only the concepts in negation normal form** (Definition 1.4.2).

Definition 1.4.2 (Negation normal form) *A concept* C *is in* negation normal form *(NNF) iff the complement constructor* ($\neg$) *only occurs in front of atomic concept symbols inside* C*.*

Any concept C can be transformed into NNF by recursively applying the rules from Table 1.1 and substituting C with C' until the negation occurs only in front of symbols of atomic concepts.

For simplicity, we can define a function nnf (Definition 1.4.3), that returns the NNF of a given concept C. It is important to note that a concept transformed into an NNF has the same meaning as the original concept.

C	C'
$\neg(D \sqcap E)$	$\neg D \sqcup \neg E$
$\neg(D \sqcup E)$	$\neg D \sqcap \neg E$
$\neg\exists R.E$	$\forall R.\neg E$
$\neg\forall R.E$	$\exists R.\neg E$

Table 1.1: Rules for transforming concept C into NNF, where D, E are concepts and R is a role.

Definition 1.4.3 (nnf(.)) *Given any concept C, we denote by* $\text{nnf}(C)$ *a concept C_{nnf} in NNF s.t. C is equivalent to C_{nnf}, i.e. $C^{\mathcal{I}} = C^{\mathcal{I}}_{nnf}$.*

1.4.2 CTree initialization and expansion

When checking the consistency of a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, the CTree is initialized based on the given ABox $\mathcal{A}$ from $\mathcal{K}$. For each unique individual a occurring in $\mathcal{A}$, a corresponding vertex a is created. To the label of the vertex a, we add all concepts C such that $a\colon C \in \mathcal{A}$. Next, for all role assertions $a, b\colon R \in \mathcal{A}$, we create an edge between the corresponding vertices a and b, and the role name R is added to the label of the edge (a, b).

After initializing the CTree, we expand it using *tableau rules* (Definition 1.4.7). Some rules are non-deterministic ($\sqcup$-rule), that is, we have more options on how to proceed. We have to try all options until we can construct a CTree without a contradiction called **clash** (Definition 1.4.4) in DL. Such a CTree is then called **clash-free** (Definition 1.4.5). If we managed to find a clash-free CTree and no rule is applicable, then the examined $\mathcal{K}$ is consistent. Otherwise, if we explore *all* options and find that they lead to a clash, then $\mathcal{K}$ is inconsistent.

The entire process of $\mathcal{K}$ consistency checking is shown in Algorithm 1.1.

Definition 1.4.4 (Clash) *There is a clash in a CTree $T = (V, E, \mathcal{L})$ iff for some $x \in V$ and for some concept C both $C \in \mathcal{L}(x)$ and $\neg C \in \mathcal{L}(x)$.*

Definition 1.4.5 (Clash-free CTree) *A CTree $T = (V, E, \mathcal{L})$ is clash-free iff none of the nodes in V contains a clash.*

Before we list *tableau rules* (Definition 1.4.7), we introduce a few more important terms which are used in rules: *ancestor*, *successor* and *R-successor* (Definition 1.4.6).

Definition 1.4.6 (Ancestor, Successor and R-successor) *Given a C-Tree $T = (V, E, \mathcal{L})$ and $x, y \in V$:*

- *x is an **ancestor** of y iff $(x, y) \in E$*
- *vice versa, y is a **successor** of x iff $(x, y) \in E$*
- *y is a **R-successor** of x iff $(x, y) \in E$ and $S \in \mathcal{L}((x, y))$ and $S \boxtimes R$, where $\boxtimes$ denotes* transitive-reflexive closure *of the role hierarchy $\sqsubseteq$, i.e. $S \boxtimes R$ iff $S = R$ or $S \sqsubseteq S_1, ..., S_n \sqsubseteq R$, where $S, S_1, ..., S_n, R \in N_R$*

The tableau rules (Definition 1.4.7) consist of an *action* to be performed and a *condition*. We apply them to the CTree vertices. Actions usually consist of adding new concepts and role names to labels or creating new vertices and edges. The rules are limited by the conditions under which they can be applied. This ensures that we cannot apply the rule indefinitely and the process eventually terminates. A special condition is that a rule can be applied to a given vertex only if the vertex is not *blocked* (Definition 1.4.8). We will explain this condition and its importance later in more detail.

Definition 1.4.7 ($\mathcal{ALCHO}$ Tableau rules) *Let $T = (V, E, \mathcal{L})$ be a CTree and C_1, C_2 be some concepts. Then the rules that are used to expand the CTree in $\mathcal{ALCHO}$ DL are defined as:*

1. **$\sqcap$-rule:** ***if*** $C_1 \sqcap C_2 \in \mathcal{L}(x)$, $x \in V$ *and* $\{C_1, C_2\} \nsubseteq \mathcal{L}(x)$ *and* x is not blocked, ***then*** $\mathcal{L}(x) := \mathcal{L}(x) \cup \{C_1, C_2\}$
2. **$\sqcup$-rule:** ***if*** $C_1 \sqcup C_2 \in \mathcal{L}(x)$, $x \in V$ *and* $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ *and* x is not blocked, ***then*** *either* $\mathcal{L}(x) := \mathcal{L}(x) \cup \{C_1\}$ *or* $\mathcal{L}(x) := \mathcal{L}(x) \cup \{C_2\}$
3. **$\exists$-rule:** ***if*** $\exists R.C \in \mathcal{L}(x)$, $x \in V$ *with no R-successor y s.t.* $C \in \mathcal{L}(y)$ *and* x is not blocked, ***then*** $V := V \cup \{z\}$, $\mathcal{L}(z) := \{C\}$ *and* $\mathcal{L}((x, z)) := \{R\}$

4. $\forall$**-rule:** ***if*** $\forall R.C \in \mathcal{L}(x)$, $x, y \in V$, y *is an* R*-successor of* x, $C \notin \mathcal{L}(y)$ *and* x is not blocked, ***then*** $\mathcal{L}(y) := \mathcal{L}(y) \cup \{C\}$

5. o**-rule:** ***if*** $\{o\} \in \mathcal{L}(x)$, $\{o\} \in \mathcal{L}(y)$, $o \in N_I$, $x \neq y$ *and* x is not blocked, ***then*** *merge* x *and* y *as follows:*

1. *for each* $(z, x) \in E$, $z \in V$, ***if*** $(z, y) \notin E$ ***then*** $\mathcal{L}((z, y)) := \mathcal{L}((z, x))$ *and* $E := E \cup \{(z, y)\} \setminus \{(z, x)\}$
2. *for each* $(z, x) \in E$, $z \in V$, ***if*** $(z, y) \in E$ ***then*** $\mathcal{L}((z, y)) := \mathcal{L}((z, y)) \cup \mathcal{L}((z, x))$ *and* $E := E \setminus \{(z, x)\}$
3. *for each* $(x, z) \in E$, $z \in V$, ***if*** $(y, z) \notin E$ ***then*** $\mathcal{L}((y, z)) := \mathcal{L}((x, z))$ *and* $E := E \cup \{(y, z)\} \setminus \{(x, z)\}$
4. *for each* $(x, z) \in E$, $z \in V$, ***if*** $(y, z) \in E$ ***then*** $\mathcal{L}((y, z)) := \mathcal{L}((y, z)) \cup \mathcal{L}((x, z))$ *and* $E := E \setminus \{(x, z)\}$
5. $\mathcal{L}(y) := \mathcal{L}(y) \cup \mathcal{L}(x)$ *and* $V := V \setminus \{x\}$

6. $\mathcal{T}$**-rule:** ***if*** $C_1 \sqsubseteq C_2 \in \mathcal{T}$, $x \in V$ *and* $\mathrm{nnf}(\neg C_1 \sqcup C_2) \notin \mathcal{L}(x)$ *and* x is not blocked, ***then*** $\mathcal{L}(x) := \mathcal{L}(x) \cup \{\mathrm{nnf}(\neg C_1 \sqcup C_2)\}$

The tableau rules depend mostly on the constructors used in the given DL. In principle, every constructor (except complement) will create a new rule. The complement constructor ($\neg$) does not have its own rule, because it is used purely for *clash detection*. The rules for a given constructor are then derived from the semantics of the constructor.

We can note that our **rules 1**–**4** are also based on constructors. However, in addition, we have **rules 5** and **6** which are based on other things contained in the $\mathcal{ALCHO}$ DL. Let us explain them in a little more detail.

o**-rule** is related to the use of *nominals*. When using nominals, a situation may arise in which 2 different vertices x, y contain the same nominal $\{o\}$, $o \in N_I$. This means that both x and y represent the individual o. Such a situation is undesirable because we require each vertex to represent exactly

one unique individual. Therefore, it is necessary to integrate these vertices into one and perform the *merge* operation described in the rules.

$\mathcal{T}$-rule ensures that we consider the *GCI axioms* $C \sqsubseteq D$, where C,D are some concepts. These axioms must also be satisfied in order for the knowledge base to be consistent. The rule is based on Lemma 1.4.1, which offers an alternative simple way to guarantee the satisfaction of the GCI axiom. The main idea is that it is enough to ensure that *each individual* either does not belong to concept C or must belong to concept D.

Lemma 1.4.1 ($\mathcal{T}$-rule) *$C \sqsubseteq D$ iff $\top \sqsubseteq \neg C \sqcup D$.*

We can note that, unlike the other rules, we can (once) apply the $\mathcal{T}$-rule to each vertex, regardless of what its label contains. Naive use of this rule, however, can lead to the constant creation of new vertices, which would cause *endless looping*. This would prevent the algorithm from terminating. A typical example of an axiom that causes this problem, if it is contained in an examined KB, is: $\mathsf{Human} \sqsubseteq \exists \mathsf{hasParent}.\mathsf{Human}$.

In order to ensure that the algorithm terminates, we apply **blocking** to vertices (Definition 1.4.8). Then, as we can see in the tableau rules, a rule can only be applied to vertices that are *not blocked*. Note that by using blocking, we will not lose any important information. Blocking guarantees that we are able to construct a final representation of the model that will contain all the necessary information.

Definition 1.4.8 (Blocking) *Given a CTree $T = (V, E, \mathcal{L})$, a node $x \in V$ is* blocked *if it has an ancestor y s.t. either $\mathcal{L}(x) \subseteq \mathcal{L}(y)$ or y is blocked.*

The whole procedure of a KB $\mathcal{K}$ consistency checking using the table algorithm is summarized in Algorithm 1.1. For a better illustration, we also provide a concrete simple example with built CTree and with the step-by-step instructions in Example 1.4.1.

Algorithm 1.1 Checking consistency of $\mathcal{K}$

Input: $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ in NNF
Output: $\mathcal{K}$ *is consistent* or $\mathcal{K}$ *is inconsistent*

function InitializeCTree($\mathcal{K}$)
    $V := \{a \mid \text{individual } a \text{ occurs in } \mathcal{A}\}$
    $\mathcal{L}(a) := \{\text{nnf}(C) \mid a\colon C \in \mathcal{A}\}$ for all $a \in V$
    $E := \{(a, b) \mid a, b\colon R \in \mathcal{A} \text{ for some role } R\}$
    $\mathcal{L}((a, b)) := \{R \mid a, b\colon R \in \mathcal{A}\}$ for all $(a, b) \in E$
    return $T = (V, E, \mathcal{L})$
end function

$T \leftarrow$ InitializeCTree($\mathcal{K}$)

while at least one tableau expansion rule is applicable **do**
    apply tableau expansion rules on T
end while

if T is clash-free **then**
    return $\mathcal{K}$ is consistent
end if
return $\mathcal{K}$ is inconsistent

Example 1.4.1 (Consistency checking of KB) *We use vocabulary from Example 1.1.1. Let $\mathcal{K}_4 = (\mathcal{T}_4, \mathcal{A}_4)$ be our DL KB, such that:*

$$\mathcal{T}_4 = \{\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}\}$$
$$\mathcal{A}_4 = \{\mathsf{fluffy}\colon \mathsf{Dog}, \mathsf{fluffy}\colon \mathsf{Good}, \mathsf{tim}, \mathsf{fluffy}\colon \mathsf{owns}\}$$

We will use the tableau algorithm and prove that KB $\mathcal{K}_4$ is consistent.

Steps:

1. *We initialize the CTree $T = (V, E, \mathcal{L})$ according to $\mathcal{A}_4$:*
   (a) $V := \{\mathsf{tim}, \mathsf{fluffy}\}$
   (b) $\mathcal{L}(\mathsf{fluffy}) := \{\mathsf{Dog}, \mathsf{Good}\}$
   (c) $E := \{(\mathsf{tim}, \mathsf{fluffy})\}$
   (d) $\mathcal{L}((\mathsf{tim}, \mathsf{fluffy})) := \{\mathsf{owns}\}$
2. *We apply $\mathcal{T}$-rule from the axiom* $\mathsf{Dog} \sqcup \mathsf{Cat} \sqsubseteq \mathsf{Pet}$ *to the vertex* tim*:* $\mathcal{L}(\mathsf{tim}) := \mathcal{L}(\mathsf{tim}) \cup \{\text{nnf}(\neg(\mathsf{Dog} \sqcup \mathsf{Cat}) \sqcup \mathsf{Pet})\} = \mathcal{L}(\mathsf{tim}) \cup \{(\neg\mathsf{Dog} \sqcap \neg\mathsf{Cat}) \sqcup \mathsf{Pet}\}$.

3. *We apply* $\sqcup$*-rule to the vertex* tim *and choose the option:* $\mathcal{L}(\mathsf{tim}) := \mathcal{L}(\mathsf{tim}) \cup \{\neg\mathsf{Dog} \sqcap \neg\mathsf{Cat}\}$.

4. *Then, we apply* $\sqcap$*-rule to the vertex* tim*:* $\mathcal{L}(\mathsf{tim}) := \mathcal{L}(\mathsf{tim}) \cup \{\neg\mathsf{Dog}, \neg\mathsf{Cat}\}$.

5. *We proceed similarly for the vertex* fluffy *and use the* $\mathcal{T}$*-rule:* $\mathcal{L}(\mathsf{fluffy}) := \mathcal{L}(\mathsf{fluffy}) \cup \{\mathrm{nnf}(\neg(\mathsf{Dog} \sqcup \mathsf{Cat}) \sqcup \mathsf{Pet})\} = \mathcal{L}(\mathsf{fluffy}) \cup \{(\neg\mathsf{Dog} \sqcap \neg\mathsf{Cat}) \sqcup \mathsf{Pet}\}$.

6. *We apply* $\sqcup$*-rule to the vertex* fluffy *and choose the option:* $\mathcal{L}(\mathsf{fluffy}) := \mathcal{L}(\mathsf{fluffy}) \cup \{\mathsf{Pet}\}$.

7. *There are no more applicable rules and constructed CTree* T *is clash-free, so the KB* $\mathcal{K}_4$ *is consistent. The resulting CTree* T *is shown in Figure 1.1.*

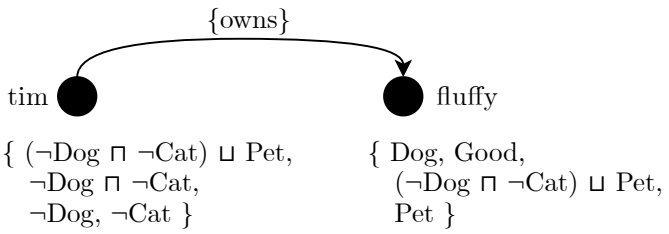


Figure 1.1: Clash-free CTree T for $\mathcal{K}_4$

In addition to proving that $\mathcal{K}_4$ *is consistent, we can also read a model* $\mathcal{M}$ *from such a clash-free CTree.*

In order to know whether a given individual i *belongs to an atomic concept* A *w.r.t.* $\mathcal{M}$ *or, on the contrary, cannot belong to it, we look at its vertex label. If its label contains* A*, then* $i^{\mathcal{M}} \in A^{\mathcal{M}}$*. If its label contains* $\neg A$*, then* $i^{\mathcal{M}} \notin A^{\mathcal{M}}$*. Otherwise, we can choose.*

In a similar way, we can determine whether a pair of individuals belong to a role.

Based on CTree T for $\mathcal{K}_4$ (Fig. 1.1), we can construct $\mathcal{M} \models \mathcal{K}_4$, for which the following applies: $\mathsf{tim}^{\mathcal{M}} \notin \mathsf{Dog}^{\mathcal{M}}$, $\mathsf{tim}^{\mathcal{M}} \notin \mathsf{Cat}^{\mathcal{M}}$, $\mathsf{fluffy}^{\mathcal{M}} \in \mathsf{Good}^{\mathcal{M}}$, $\mathsf{fluffy}^{\mathcal{M}} \in \mathsf{Dog}^{\mathcal{M}}$, $\mathsf{fluffy}^{\mathcal{M}} \in \mathsf{Pet}^{\mathcal{M}}$ *and* $(\mathsf{tim}^{\mathcal{M}}, \mathsf{fluffy}^{\mathcal{M}}) \in \mathsf{owns}^{\mathcal{M}}$.

1.5 Ontologies

Decidable DLs are used as a representational formalism for ontologies [21, 8] and to enable reasoning over them. In this section, we will briefly discuss what ontology is, as this concept will be important in the implementation parts.

The concept of ontology has different meanings in different fields. In Computer Science, an ontology represents a knowledge structure. It is defined as a *shared formal conceptualization of a domain* [8]. That means that it represents a common understanding of a domain that can be shared by a certain community of experts. It uses precise, unambiguous language and it defines the entities and the relationships between them within a certain application domain.

We will mainly deal with OWL ontologies [3] that are defined using OWL language, which is based on DLs. In the context of this work, we will consider ontologies as structured files that capture knowledge about a certain domain. Therefore, we will use ontology as a representation of a knowledge base.

Some different terms are used in the context of ontology than in DLs. We introduce ontology terms and their counterpart in DLs [21] using Table 1.2.

DLs term	**ontology term**
knowledge base	ontology
concept	class
role	object property
–	data property
–	data type
axiom	axiom
vocabulary	vocabulary

Table 1.2: Terms in DLs and corresponding ontology terms.

OWL ontologies can work and reason with data types, e.g. boolean, integer or string. In Table 1.2, we can see the ontology entity named *data property*, which represents the relationship between an individual and a data type.

We will continue to use the mentioned DL terms in the implementation context. However, it is important to point out that ontologies have some features that are not included in DLs, for example, working with data types. This will prove to be important, so we wanted to highlight these differences in this chapter.

Chapter 2

Abduction

In reasoning, we can encounter 3 basic types of inference [16, 19, 7] or, in other words, ways of deriving new knowledge from known background knowledge: *deduction*, *induction* and *abduction*. *Deduction* consists in drawing logical consequences which follow from background knowledge. *Induction* consists in deriving general conclusions (intensional knowledge) which explain specific facts (extensional knowledge). **Abduction** consists in searching for an explanation of some observation. It can be seen as backward reasoning from an observation back to its cause. For example, from the given evidence, we want to obtain who the killer was or from symptoms, we want to derive a diagnosis that causes them.

The conclusions of deduction are generally true with regard to background knowledge. Induction and abduction, on the other hand, do not guarantee their conclusions to be true (only *probably* true). However, they can provide more new knowledge than can be obtained by deduction.

In our work, we focus on the third type of inference – *abduction*. In this chapter, we will define the abductive reasoning task we will be dealing with and look at some methods that can solve the given task.

2.1 Abduction problem

There are several abductive reasoning tasks [7, 17, 11], but we will specifically focus on solving one of them called the *ABox abduction problem* (Definition 2.1.1).

Given an observation $\mathcal{O}$ and KB $\mathcal{K}$, our goal is to find an explanation $\mathcal{E}$ that together with $\mathcal{K}$ entails $\mathcal{O}$. In other words, we want to find out why $\mathcal{O}$ does not follow from $\mathcal{K}$ or what is missing for $\mathcal{O}$ to follow from $\mathcal{K}$.

The problem is called the *ABox abduction* [7] because observations and explanations can consist only of *assertions from ABox*.

Definition 2.1.1 (ABox abduction problem) *An ABox abduction problem is a pair $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ such that $\mathcal{K}$ is a DL KB, $\mathcal{O}$ is an ABox assertion called* ***observation****. A solution of $\mathcal{P}$, also called* ***explanation****, is a finite set of ABox assertions $\mathcal{E}$ such that $\mathcal{K} \cup \mathcal{E} \models \mathcal{O}$.*

In order for solutions to the abduction problem to be meaningful, abduction is often limited and the solution (or explanation) must have certain properties to be considered valid. There are several properties that can be required from a solution. The most typical and also ones we use are *consistency*, *relevance*, *explanatoriness* and *minimality*. Solutions with these properties will be called **desired** (Definition 2.1.2).

Definition 2.1.2 (Desired explanation) *An explanation $\mathcal{E}$ of an abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ is called* desired *if it has these properties:*

- *$\mathcal{E}$ is* ***consistent*** *iff $\mathcal{E} \cup \mathcal{K}$ is consistent*
- *$\mathcal{E}$ is* ***relevant*** *iff $\mathcal{E} \not\models \mathcal{O}$, i.e. $\mathcal{E}$ does not entail $\mathcal{O}$*
- *$\mathcal{E}$ is* ***explanatory*** *iff $\mathcal{K} \not\models \mathcal{O}$, i.e. $\mathcal{K}$ does not entail $\mathcal{O}$*
- *$\mathcal{E}$ is syntactically* ***minimal*** *iff there is no other solution $\mathcal{E}'$ of $\mathcal{P}$ such that $\mathcal{E}' \subseteq \mathcal{E}$.*

The most important property of the desired explanation is *consistency*. As we mentioned in the previous chapter, if $\mathcal{K} \cup \mathcal{E}$ is inconsistent, any observation would follow from it. This would produce explanations which make no sense.

Relevance and *explanatoriness* ensure that the solution does not depend purely on the knowledge base or only on the explanation itself, but on both of them.

In general, an abductive problem can have an infinite number of solutions. These solutions are not equally important, and *minimality* expresses our preference for one solution over another.

2.1.1 Observation types

An observation is not limited and thus can be *any* ABox assertion. However, for the sake of clarity, it is necessary to distinguish between the types of observations that can occur in the ABox abduction problem.

Single observation is an observation that consists of *one ABox assertion*. According to the type of ABox assertion, we can further distinguish between *atomic* and *complex* single observation.

Atomic observation is a single observation that consists of *atomic* concept (or role) assertion.

Complex observation is a single observation that consists of a *complex* concept assertion.

Multiple observation is an observation that consists of *multiple ABox assertions*.

There are two ways how we can solve the ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ with a multiple observation $\mathcal{O}$ [18]. We can either *divide the problem* $\mathcal{P}$ into n subproblems $\mathcal{P}_i = (\mathcal{K}, \mathcal{O}_i)$, where $\mathcal{O}_i \in \mathcal{O}$. Then we can solve $\mathcal{P}_i$ individually as ABox problems with a single observation and combine results to obtain a solution for the original problem $\mathcal{P}$. Or we can also use

the second, more efficient method, and *reduce the multiple observation* to a single observation according to Lemma 2.1.1. Then, it is enough to solve the ABox abduction problem $\mathcal{P}'$ with a single observation. For more details and proof of this statement, you can refer to the work of Pukancová [19].

Lemma 2.1.1 (Multiple observation reduction) *Let $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ be an ABox abduction problem with a multiple observation $\mathcal{O} = \{a\colon C_1, \ldots, a_n\colon C_n, b_1, c_1\colon R_1, \ldots, b_m, c_m\colon R_m, d_1, e_1\colon \neg S_1, \ldots, d_p, e_p\colon \neg S_p\}$, where C_i are concepts, $R_i, S_j \in N_R$ and $a_i, b_j, c_j, d_k, e_k \in N_I$ for $i \in [1, n]$, $j \in [1, m]$ and $k \in [1, p]$. Let $\mathcal{E} \subseteq \{a\colon A,\ a\colon \neg A,\ a, b\colon R,\ a, b\colon \neg R \mid A \in N_C,\ R \in N_R,\ a, b \in N_I\}$. Let $\mathcal{P}' = (\mathcal{K}, \mathcal{O}')$ be an ABox abduction problem with a single observation $\mathcal{O}' = s_0\colon X$, such that s_0 is a new individual w.r.t. $\mathcal{K}, \mathcal{O}$ and $\mathcal{E}$, and*

$$\begin{aligned} X = &(\neg\{a_1\} \sqcup C_1) \sqcap \ldots \sqcap (\neg\{a_n\} \sqcup C_n) \\ &\sqcap (\neg\{b_1\} \sqcup \exists R_1.\{c_1\}) \sqcap \ldots \sqcap (\neg\{b_m\} \sqcup \exists R_m.\{c_m\}) \\ &\sqcap (\neg\{d_1\} \sqcup \forall S_1.\neg\{e_1\}) \sqcap \ldots \sqcap (\neg\{d_p\} \sqcup \forall S_p.\{e_p\}) \end{aligned}$$

Then $\mathcal{E}$ is an explanation of $\mathcal{P}$ if and only if it is an explanation of $\mathcal{P}'$.

You can notice that s_0 cannot appear in the explanations, otherwise, $\mathcal{P}$ and $\mathcal{P}'$ would not have the same solutions.

2.1.2 Abducibles

According to Definition 2.1.1, an explanation of the ABox abduction problem can consist of arbitrary ABox assertions. However, this means that we have an *infinite number* of concept expressions from which we can construct an explanation. Therefore only some meaningful finite subset of assertions is typically considered. We call this subset **abducibles** and denote it Abd. Hence we can define a new version of the ABox abduction problem (Definition 2.1.3), where we take into account the set of abducibles. An explanation of such a problem can only consist of assertions from abducibles.

Definition 2.1.3 (ABox abduction problem with abducibles) *Let* Abd *be a finite set of ABox assertions. An ABox abduction problem is a pair* $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ *such that* $\mathcal{K}$ *is a DL KB,* $\mathcal{O}$ *is an ABox assertion called* ***observation****. A solution of* $\mathcal{P}$*, also called* ***explanation****, is a finite set of ABox assertions* $\mathcal{E} \subseteq$ Abd *such that* $\mathcal{K} \cup \mathcal{E} \models \mathcal{O}$*.*

In our work, we limit explanations to only atomic and negated atomic assertions, so Abd $\subseteq \{a\colon A,\ a\colon \neg A \mid A \in N_C, a \in N_I\} \cup \{a, b\colon R,\ a, b\colon \neg R \mid R \in N_R, a, b \in N_I\}$.

An example of an ABox abduction problem along with its solutions is shown in the Example 2.1.1.

Example 2.1.1 *Let us model the situation of Tim's dog Fluffy. Tim found out that Fluffy is sad and wants to find out why.*

To save space, assume that we have defined the vocabulary. Let us have a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ *such that:*

$$\mathcal{T} = \{\mathsf{Alone} \sqcap \mathsf{WantsToPlay} \sqsubseteq \mathsf{Sad},\ \mathsf{Hungry} \sqsubseteq \mathsf{Sad}\}$$
$$\mathcal{A} = \{\}$$

Observation is $\mathcal{O} = \{\mathsf{fluffy}\colon \mathit{Sad}\}$.

We will look for explanations only in atomic and negated atomic assertions. Then all possible (desired) solutions to this ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ *are:*

$\mathcal{E}_1 = \{\mathsf{fluffy}\colon \mathsf{Hungry}\}$

$\mathcal{E}_2 = \{\mathsf{fluffy}\colon \mathsf{Alone}, \mathsf{fluffy}\colon \mathsf{WantsToPlay}\}$

2.2 Methods for searching explanations

In the first part of this chapter, we presented the ABox abduction problem. In the second part, we will show two methods by which we can solve it and find all its possible explanations (within the set of abducibles). Such methods,

which guarantee finding all explanations, are called *complete*. First, we will get acquainted with the well-known MHS algorithm [20, 17], and then we will present the MHS-MXP algorithm [11, 10].

Before introducing the methods, let us simplify the problem according to Lemmata 1.3.1 and 1.3.2. Based on this, we can transform the entailment problem $\mathcal{K} \cup \mathcal{E} \models \mathcal{O}$ into a consistency checking of $\mathcal{K} \cup \mathcal{E} \cup \{\neg\mathcal{O}\}$. Our goal is to find such explanations $\mathcal{E}$ that $\mathcal{K} \cup \mathcal{E} \cup \{\neg\mathcal{O}\}$ will be inconsistent.

Note that we can also apply this to a multiple observation, which can be transformed to a single observation after applying the reduction from Lemma 2.1.1.

2.2.1 MHS algorithm

The basis of the method is **Minimal Hitting Set (MHS) algorithm** [20], which was later adapted to solve the ABox abduction problem [17, 19, 11]. First, we will briefly explain the fundamental terms and the way the basic MHS algorithm works, and then we will describe how we can use it to find explanations.

Basic MHS algorithm

The basic MHS algorithm is used to search for **minimal hitting sets** (Definition 2.2.1). In short, if we have a collection of sets, then their hitting set is the set that contains an element from *each of them*. Such a set is minimal if there is no proper subset which is also the hitting set of the given collection of sets. For example, we can have a collection of sets $\{\{1, 2, \mathbf{3}\}\, \{\mathbf{3}, 4\}, \{\mathbf{5}, 6\}\}$, then one of their minimal hitting set is $\{\mathbf{3,5}\}$.

Definition 2.2.1 (Minimal hitting set) *A hitting set for a collection of sets F is a set $H \cap S \neq \emptyset$ for every $S \in F$. A hitting set H for F is* minimal *if there is no other hitting set H' for F s.t. $H' \subsetneq H$.*

Minimal hitting sets can be found by constructing a **pruned HS-tree** (Definition 2.2.2) by breadth-first search (BFS). Optimizations such as the

breadth-first search approach and *pruning* make the algorithm more efficient and ensure that the found hitting sets are truly minimal.

Definition 2.2.2 (Pruned HS-tree) *A HS-tree for F is $T = (V, E, L, H)$, where (V, E) is a minimal tree constructed by BFS in which the labelling function L labels the nodes of V by elements of F, the edges of E by elements of sets in F, and $H(n)$ is the set of edge labels from the root node to $n \in V$, s.t.*

1. *for the root $r \in V$: $L(r) = S$ for some $S \in F$, if $F \neq \emptyset$, otherwise $L(r) = \emptyset$*
2. *for each* not pruned *$n \in V$: $L(n) = S$ for some $S \in F$ s.t. $S \cap H(n) = \emptyset$, if such S exists, otherwise $L(n) = \emptyset$*
3. *for each* pruned *$n \in V$: $L(n) = \times$*
4. *each* not pruned *$n \in V$ has a successor n_σ for each $\sigma \in L(n)$ with $L(n, n_\sigma) = \sigma$*

Node $n \in V$ ***is pruned*** *if there is another node n' such that:*

- *either $H(n') \subseteq H(n)$ and $L(n') = \emptyset$*
- *or $H(n') = H(n)$ and $L(n') = S \in F$*

After creating such an HS-tree T for a given collection F, we can obtain hitting sets F from the edge labels of tree branches that were not pruned (Reiter's theorem 2.2.1).

Theorem 2.2.1 (Reiter's theorem) *Let $T = (V, E, L, H)$ be a pruned HS-tree for a collection of sets F. Then $\{H(n) \mid n \in V,\ L(n) = \emptyset$ and n is not pruned $\}$ is the collection of all minimal sets for F.*

The use of the basic MHS algorithm on a collection of sets of numbers is shown in Example 2.2.1.

Example 2.2.1 (Searching for minimal hitting sets) *Let us have collection of sets* $F = \{\{1,2,3\}\{3,4\},\{5,6\}\}$. *For the given* F *we can create a pruned HS-tree (Fig. 2.1).*

Note that the HS-tree for F *can also look different since we are choosing a set to label a node. For example, we could start constructing a tree with a root labelled with set* $\{3,4\}$ *or* $\{5,6\}$. *In the end, however, we always end up with the same hitting sets for* F.

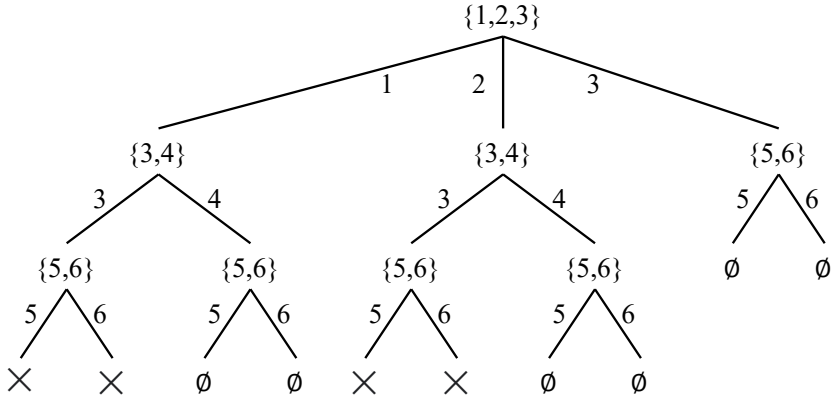


Figure 2.1: Pruned HS-tree for $F = \{\{1,2,3\}\{3,4\},\{5,6\}\}$

After constructing the tree, we look at the leaves labelled with $\emptyset$. *The branch corresponding to such a leaf contains a minimal hitting set, which is obtained by uniting the edge labels of the given branch. The resulting minimal hitting sets of* F*:* $\{1,4,5\}$, $\{1,4,6\}$, $\{2,4,5\}$, $\{2,4,6\}$, $\{3,5\}$, $\{6,5\}$.

Main idea of explanations searching with the MHS algorithm

As we have already mentioned, when solving a problem $\mathcal{P} = (\mathcal{K},\mathcal{O})$ we try to find $\mathcal{E}$ such that $\mathcal{K} \cup \mathcal{E} \cup \{\neg\mathcal{O}\}$ is inconsistent, i.e. it has no models. We can therefore consider all $\mathcal{K} \cup \{\neg\mathcal{O}\}$ models and try to eliminate each of them using $\mathcal{E}$. This can be done by constructing $\mathcal{E}$ to contain one *negated* assertion from each model of $\mathcal{K} \cup \{\neg\mathcal{O}\}$. Therefore, $\mathcal{E}$ is basically the minimal hitting set of the collection of $\mathcal{K} \cup \{\neg\mathcal{O}\}$ models with *negated* assertions.

However, we have yet to consider the set of abducibles, denoted Abd,

since we want the explanations of the problem to consist only of them. It turned out that this can be achieved simply by excluding assertions that are not in abducibles during the search (Observation 2.2.1).

Observation 2.2.1 *The minimal explanations of* $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ *on* Abd *directly corresponds to the minimal hitting sets of* $\{\text{Abd}(M) \mid M \models \mathcal{K} \cup \neg\{\mathcal{O}\}\}$ *where* $\text{Abd}(M) = \{\phi \in \text{Abd} \mid M \not\models \phi\}$.

Note that $\text{Abd}(M)$ already contains assertions that do not follow from M, negated M assertions. We have to consider one more situation, when $\text{Abd}(M) = \emptyset$ for some $M \models \mathcal{K} \cup \{\neg\mathcal{O}\}$. In such a case, a problem $\mathcal{P}$ has no explanation on Abd (Observation 2.2.2).

Observation 2.2.2 *If* $\text{Abd}(M) = \emptyset$ *for some* $M \models \mathcal{K} \cup \{\neg\mathcal{O}\}$, *then* $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ *has no explanation on* Abd.

So we can use the MHS algorithm to find all minimal explanations of an ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ by finding minimal hitting sets of $\{\text{Abd}(M) \mid M \models \mathcal{K} \cup \{\neg\mathcal{O}\}\}$.

Explanations searching with the MHS algorithm

Before presenting the algorithm itself, it is important to point out one difference regarding collection F. As we can notice, in the previous examples of MHS, it is assumed that we already know collection F at the beginning. In this case, it would be necessary to get all the models in the preprocessing before starting the MHS algorithm itself. However, it is possible and more optimal to modify the algorithm and obtain these models gradually during its operation. So, in the resulting algorithm, we will use this way of obtaining models.

The MHS algorithm adapted to the search for explanations (Algorithm 2.1) receives KB $\mathcal{K}$, observation $\mathcal{O}$ and a set of abducibles Abd and returns a set of explanations $\mathcal{S}_\mathcal{E}$ that solve $\mathcal{P} = (\mathcal{K}, \mathcal{O})$. The first step is to check if $\mathcal{K} \cup \{\neg\mathcal{O}\}$ is consistent (line 1). If it is not, it means that $\mathcal{O}$ already

follows from KB $\mathcal{K}$ and there is *nothing to explain* (explanations would not be explanatory) and we are done (line 3). If it is consistent, we obtain the finite representation M of the model by extracting atomic and negated atomic assertions from the real model $\mathcal{I}$.

$$\begin{aligned} M = \{&a\colon C \mid \mathcal{I} \models a\colon C, C \in \{A, \neg A\}, A \in N_C, a \in N_I\} \\ &\cup \{a, b\colon R \mid \mathcal{I} \models a, b\colon R, R \in N_R, a, b \in N_I\} \\ &\cup \{a, b\colon \neg R \mid \mathcal{I} \models a, b\colon \neg R, R \in N_R, a, b \in N_I\} \end{aligned}$$

We will work with these finite representations and call them *models*.

Next, we initialize the HS-tree T, which root r we label with $\text{Abd}(M)$, where M is the obtained model from the first step (line 5). Subsequently, we construct T in a similar way as was stated in the basic MHS algorithm (Definition 2.2.2). For each assertion σ in $\text{Abd}(M)$, we create the successor n_σ and the edge (r, n_σ) will be labelled by the given assertion σ (line 6).

We then go through the successors and process them (line 8). For each node, we check if the node should be pruned (line 9). If it is not pruned, we have to label it. Since we calculate models on the fly, to obtain M, we have to call the consistency check of $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{H}(n)$. If it is consistent, we label the node with $\text{Abd}(M)$, where M is the obtained model (line 12) and we can create the successors of the given node (line 16). If $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{H}(n)$ is inconsistent, $\mathcal{H}(n)$ is our potential explanation. However, since we have the properties that we require from the explanation to be desired (Definition 2.1.2), we still have to check *relevance* and *consistency* (line 13). Note that other properties are already ensured. *Minimality* is ensured by the MHS algorithm itself and *explanatoriness* was checked in the first step. If the explanation has the given properties, we can store it in the set of explanations $\mathcal{S}_\mathcal{E}$ (line 14). Otherwise, the node is pruned. We continue with the construction of the T as long as we have nodes to process.

In Figure 2.2, we can see the HS-tree constructed for the previous Example 2.1.1 of the ABox abduction problem, where we looked for explanations

Algorithm 2.1 MHS($\mathcal{K}$,O,Abd)

Input: Knowledge base $\mathcal{K}$, observation O, abducibles Abd
Output: $\mathcal{S}_\mathcal{E}$ all explanations of $\mathcal{P} = (\mathcal{K}, O)$ w.r.t. Abd

$M \leftarrow$ a model M of $\mathcal{K} \cup \{\neg O\}$
if $M =$ `null` **then**
    return `"nothing to explain"`
end if
$T \leftarrow (V = \{r\}, E = \emptyset, L = \{r \mapsto \text{Abd}(M)\})$
for each $\sigma \in L(r)$ create new σ-successor n_σ of r
$\mathcal{S}_\mathcal{E} \leftarrow \{\}$
while exists next node n in T w.r.t. BFS **do**
    if n can be pruned **then**
        prune n
    else if exists model M of $\mathcal{K} \cup \{\neg O\} \cup H(n)$ **then**
        label n by $L(n) \leftarrow \text{Abd}(M)$
    else if $H(n)$ is desired **then**
        $\mathcal{S}_\mathcal{E} \leftarrow \mathcal{S}_\mathcal{E} \cup \{H(n)\}$
    end if
    for each $\sigma \in L(n)$ create new σ-successor n_σ of n
end while
return $\mathcal{S}_\mathcal{E}$

of why Fluffy was sad.

Properties of the MHS algorithm: The advantage of the MHS algorithm is that it can be used for DL with arbitrary expressivity. Another advantage is that the reasoner is used as a black box, so any DL reasoner using the tableau algorithm that can perform a consistency check and from which a model can be extracted, may be used.

We can notice that in the HS-tree depth n, we find explanations of size n (the number of assertions it consists of) and that we already have all smaller explanations. Note that thanks to BFS, the search for explanations can be limited by max explanation size. If we want to find explanations up to size n, we just need to construct an HS-tree to depth n.

The MHS algorithm is *very inefficient* complete method. Especially in the case when there is at least one long explanation, it is inappropriate to use it, because in order to find an explanation of length n, the HS-tree must be constructed to depth n.

Moreover, the method cannot determine whether it has already found all explanations and must continue building the HS-tree.

2.2.2 Hybrid algorithm

Due to MHS algorithm inefficiency, other methods are being investigated to solve an ABox abduction problem $\mathcal{P}$. This algorithm is inspired by *incomplete* methods, which are not guaranteed to find all explanations, but are able to find *multiple* explanations very efficiently. Such a method is, for example, MergeXplain (MXP) algorithm [22]. The main idea of this algorithm is the combination of the MHS algorithm, which navigates the search through the explanations space, and the MXP, which task is to increase the efficiency of obtaining explanations.

In this section, we will briefly explain how the MXP algorithm works, then we will describe how it is integrated into the MHS algorithm and present the resulting hybrid algorithm called MHS-MXP.

MergeXplain algorithm

The **MergeXplain (MXP) algorithm** [22] was originally designed to find minimal inconsistent subsets, called *conflicts*, of some overconstrained knowledge base $\mathcal{K} = \mathcal{B} \cup \mathcal{C}$, where $\mathcal{B}$ is considered to be the correct consistent background theory and $\mathcal{C}$ is the "suspicious" part in which we are looking for conflicts. During one algorithm run, *multiple minimal conflicts* can be found, but it does not guarantee to find all of them.

MXP uses a *divide-and-conquer* strategy to find conflicts. It divides the problem into several smaller subproblems and then merges them together reconstructing some of the minimal conflicts. MXP requires only basic reasoning functionality such as consistency checking.

MXP (Algorithm 2.2) receives $\mathcal{B}$ and $\mathcal{C}$, and returns a set of minimal conflicts Γ. In the beginning, two checks are made to decide whether it makes sense to look for conflicts at all (line 1–5). If $\mathcal{B}$ is not consistent, then it contradicts our assumption that $\mathcal{B}$ *should be consistent*. If, on the

other hand, $\mathcal{B} \cup \mathcal{C}$ is consistent, there are no conflicts. In any other case, the conflicts are there and we search for them by calling the recursive function `findConflicts`.

Function `findConflicts` again receives $\mathcal{B}$ and $\mathcal{C}$, and returns the pair $\mathcal{C}'$ and Γ, where $\mathcal{C}'$ is a maximal subset of $\mathcal{C}$ s.t. $\mathcal{B} \cup \mathcal{C}'$ is consistent and Γ is a *set of minimal conflicts*. In the first part of the function (line 9–17), $\mathcal{C}$ is divided into $\mathcal{C}_1$ and $\mathcal{C}_2$, and we search for a conflict *independently* in each part. In the end, we save all found conflicts in Γ. In the second part (line 18–23) of the function, we search for lost conflicts that were divided during the splitting (one part of them is in $\mathcal{C}_1$ and the other in $\mathcal{C}_2$). We are able to recover some of them using the function `getConflict`.

Function `getConflict` receives $\mathcal{B}$, $\mathcal{C}$ and an auxiliary parameter $\mathcal{D}$ and returns one minimal conflict. The function proceeds in a similar way as the first part of `findConflicts`: a minimal conflict is drawn from $\mathcal{C}$, which is recursively divided into $\mathcal{C}_1$ and $\mathcal{C}_2$. The main difference is that it finds *only one* minimal conflict and sets $\mathcal{C}_1$ and $\mathcal{C}_2$ are *not independent*. We can see that, unlike `findConflicts`, in `getConflict` the result $\mathcal{D}_2$ of the first recursive call (line 33) is used in the second recursive call (line 34) and one minimal conflict is created by uniting the results from both calls (line 35).

After obtaining the minimal conflict γ using `getConflict`, we insert it into the resulting set of minimal conflicts Γ (line 22). We also need to remove one axiom $\sigma \in \gamma$ from $\mathcal{C}_1$ or $\mathcal{C}_2$ to avoid infinite looping (line 21).

Explanation searching with the MXP algorithm: MXP can be directly used to solve the ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ with abducibles Abd. By calling MXP($\mathcal{K} \cup \{\neg\mathcal{O}\}$, Abd) we are able to get *multiple* minimal explanations (Theorem 2.2.2). More details can be found in the work of Shchekotykhin et al. [22]. However, it is important to note that we might obtain explanations which are not desired.

Theorem 2.2.2 *Assume an ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ and a set of abducibles* Abd. *If $\mathcal{P}$ has at least one explanation $\gamma \subseteq$ Abd then calling*

Algorithm 2.2 MXP($\mathcal{B}$,$\mathcal{C}$)

Input: background theory $\mathcal{B}$, set of possibly faulty constraints $\mathcal{C}$
Output: a set of minimal conflicts Γ

```
1: if ¬isConsistent(B) then
2:     return "no explanation"
3: else if isConsistent(B ∪ C) then
4:     return ∅
5: end if
6: ⟨_, Γ⟩ ← FindConflicts(B, C)
7: return Γ

8: function FindConflicts(B, C)
9:     if isConsistent(B ∪ C) then
10:        return ⟨C, ∅⟩
11:    else if |C| = 1 then
12:        return ⟨∅, {C}⟩
13:    end if
14:    Split C into disjoint, non-empty sets C1 and C2
15:    ⟨C1', Γ1⟩ ← FindConflicts(B, C1)
16:    ⟨C2', Γ2⟩ ← FindConflicts(B, C2)
17:    Γ ← Γ1 ∪ Γ2
18:    while ¬isConsistent(C1' ∪ C2' ∪ B) do
19:        X ← GetConflict(B ∪ C2', C2', C1')
20:        γ ← X ∪ GetConflict(B ∪ X, X, C2')
21:        C1' ← C1'\{σ} where σ ∈ X
22:        Γ ← Γ ∪ {γ}
23:    end while
24:    return ⟨C1' ∪ C2', Γ⟩
25: end function

26: function GetConflict(B, D, C)
27:    if D ≠ ∅ ∧ ¬isConsistent(B) then
28:        return ∅
29:    else if |C| = 1 then
30:        return C
31:    end if
32:    Split C into disjoint, non-empty sets C1 and C2
33:    D2 ← GetConflict(B ∪ C1, C1, C2)
34:    D1 ← GetConflict(B ∪ D2, D2, C1)
35:    return D1 ∪ D2
36: end function
```

MXP($\mathcal{K} \cup \{\neg\mathcal{O}\}$, Abd) *returns a non-empty set* Γ *of minimal explanations of* $\mathcal{P}$.

For a better understanding, we show an example of an MXP call in Example 2.2.2.

Example 2.2.2 (MXP call) *Let* $\mathcal{K} = \{A \sqcap B \sqsubseteq D, C \sqsubseteq D\}$, $\mathcal{O} = \{a\colon D\}$ *and* $\mathrm{Abd} = \{a\colon A, a\colon B, a\colon C\}$. *Then the ABox abduction problem* $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ *has two minimal explanations:* $\{a\colon A, a\colon B\}$ *and* $\{a\colon C\}$.

$\mathrm{MXP}(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathrm{Abd})$ *is called. The initial conditions are not met and so the function* `findConflicts`$(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathrm{Abd})$ *is called. Subsequently,* $\mathcal{C}$ *is divided into* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *in some way. In our case, let* $\mathcal{C}_1 = \{a\colon A, a\colon B\}$ *and* $\mathcal{C}_2 = \{a\colon C\}$.

> `findConflicts`$(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathcal{C}_1)$ *is called.* $\mathcal{C}_1$ *does not meet the conditions, so it is further divided into* $\mathcal{C}_{11} = \{a\colon A\}$ *and* $\mathcal{C}_{12} = \{a\colon B\}$.
>
> > `findConflicts`$(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathcal{C}_{11})$ *is called.* $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{C}_{11}$ *is consistent, so* $< \mathcal{C}_{11}, \emptyset >$ *is returned.*
> >
> > `findConflicts`$(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathcal{C}_{12})$ *is called.* $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{C}_{12}$ *is consistent, so* $< \mathcal{C}_{12}, \emptyset >$ *is returned.*
> >
> > *No conflicts have been found, so we are not adding anything to* Γ *yet. Next, the while loop is entered, because* $\mathcal{C}_{11} \cup \mathcal{C}_{12} \cup \mathcal{K} \cup \{\neg\mathcal{O}\}$ *is inconsistent.*
> >
> > `getConflict`$(\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{C}_{12}, \mathcal{C}_{12}, \mathcal{C}_{11})$ *is called. Since* $\mathcal{C}_{11}$ *has size 1, it is returned and so* $X = \mathcal{C}_{11}$.
> >
> > `getConflict`$(\mathcal{K} \cup \{\neg\mathcal{O}\} \cup X, X, \mathcal{C}_{12})$ *is called. Similarly,* $\mathcal{C}_{12}$ *has size 1,* $\mathcal{C}_{12}$ *is returned and so* $\gamma = X \cup \mathcal{C}_{12} = \{a\colon A, a\colon B\}$.
> >
> > *Next, we remove* $\{a\colon A\}$ *from* $\mathcal{C}_{11}$*, so* $\mathcal{C}_{11} = \emptyset$. γ *is then added to the set of conflicts* $\Gamma = \{\{a\colon A, a\colon B\}\}$. $\mathcal{C}_{11} \cup \mathcal{C}_{12} \cup \mathcal{K} \cup \{\neg\mathcal{O}\}$ *is consistent and the while loop is broken out.* $< \mathcal{C}_{12}, \Gamma >$ *is returned.*
>
> `findConflicts`$(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathcal{C}_2)$ *is called.* $\mathcal{C}_2$ *is a conflict of size 1, so it passes the second condition and returns* $< \emptyset, \{\mathcal{C}_2\} >$.

$\Gamma = \{\{a\colon A, a\colon B\}, \{a\colon C\}\}$. *While loop is not entered because* $\mathcal{C}_{12} \cup \emptyset \cup \mathcal{K} \cup \{\neg\mathcal{O}\}$ *is consistent. Therefore MXP call returns* $\Gamma = \{\{a\colon A, a\colon B\}, \{a\colon C\}\}$*, which means that in this case both minimal explanations were found.*

Properties of the MXP algorithm: We have mentioned that with the help of MXP, we can find *multiple minimal explanations*. Moreover, MXP can find them fast and efficiently. However, it is an *incomplete* method and thus does not guarantee that all explanations will be found. This happens mainly in the case of overlapping explanations. An example of such a situation can be found in the work of Homola et al. [10].

However, MXP also has other interesting properties that were not mentioned yet. MXP *always finds all explanations of size 1* (Observation 2.2.3) [11]. This is guaranteed by a way of dividing $\mathcal{C}$, which is divided unless the subset is consistent or has size 1 (line 9–12).

Observation 2.2.3 *Given an ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$, a set of abducibles* Abd*, and any $\gamma \subseteq$ Abd s.t. $|\gamma| = 1$, if $\mathcal{K} \cup \gamma \models \mathcal{O}$ then $\gamma \in \mathrm{MXP}(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathrm{Abd})$.*

Another property is that if during the MXP algorithm run, we do not find any explanation with a size greater than 1, then such an explanation does not exist (Theorem 2.2.3) [11]. In this case, we can be sure that we got *all minimal explanations from one MXP run*. This property turns out to be very useful for the combined MHS-MXP algorithm.

Theorem 2.2.3 *Given an ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$, a set of abducibles* Abd*, let $\Gamma = \mathrm{MXP}(\mathcal{K} \cup \{\neg\mathcal{O}\}, \mathrm{Abd})$. If there is no $\gamma \in \Gamma$ s.t. $|\gamma| > 1$, then for all minimal $\delta \subseteq$ Abd s.t. $\mathcal{K} \cup \delta \models \mathcal{O}$ we have that $\delta \in \Gamma$.*

The last key observation regarding MXP is that with a *slight modification of inputs*, the space of conflicts can be divided differently and thus we can get a *different set of explanations* than in the previous calls. More details and a sketch of the proof are available in the work of Homola et al. [11].

MHS-MXP algorithm

As we mentioned, MHS is a *complete* method that is very *inefficient*. On the other hand, MXP is *efficient* but *incomplete*. Therefore, neither method

is ideal on its own. However, we could try to take advantage of the MXP property, that with a slight change of inputs, we can get a different set of explanations and thus gradually get them all with multiple MXP calls. There are more ways how this can be done. We could use a naive approach and randomly divide the conflict space in different ways. However, this comes with a major problem that we would not know when to stop. Another approach is to use a systematic search from the MHS method and conduct a space search by constructing an HS-tree. The MHS-MXP algorithm [11, 10] is based on the second approach, which, as we will see later, makes good use of the MXP algorithm properties.

The resulting MHS-MXP algorithm (Algorithm 2.3) is very similar to the MHS algorithm (Algorithm 2.1). The main difference is that in each node, instead of the consistency check, MXP is used by calling the function `findConflicts`. Note that the consistency check is still used in the initial condition and also within `findConflicts`. The consistency check (line 22) was slightly modified and it also stores the extracted models in the set Mod now. Similar to the previous cases, we will go through the algorithm and briefly explain it.

The input and output of the MHS-MXP algorithm are identical to those of the MHS algorithm. In the beginning, we initialize the set of conflicts Con, which will contain explanations and the set of models Mod.

Initial conditions are performed to decide whether it is meaningful to look for explanations for the given input. Similar to the case of MHS, we will try to obtain a model of $\mathcal{K} \cup \{\neg O\}$ through the consistency check (line 3). If a model exists it is stored into Mod. Otherwise, there is *nothing to explain* and algorithm terminates (line 4).

The second initial condition (line 5) is based on Observation 2.2.2. So if $\text{Abd}(M) = \emptyset$, where M is the model obtained from the consistency check, then the problem has no explanations on the given Abd set and the algorithm terminates.

Subsequently, we initialize the HS-tree T with the root r (line 8). Note

that, unlike MHS, r is not labelled yet and thus also its successors are not created.

Next, we process the nodes of T (including root r). Each node is processed as follows. First, we check if it should be pruned (line 10) as we did in MHS. If it is not pruned, we call the function `findConflicts`, which receives $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{H}(n)$ as the correct background theory and $\text{Abd} \setminus \mathcal{H}(n)$ as the "suspicious" part (line 13). We basically search the space of explanations that contain $\mathcal{H}(n)$. We can notice that based on $\mathcal{H}(n)$, we slightly change the inputs and thus have a chance to receive new explanations.

The function `findConflict` finds a set of conflicts Γ. However, these conflicts are conflicts with respect to the modified knowledge base $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{H}(n)$. Since we want conflicts with respect to $\mathcal{K} \cup \{\neg\mathcal{O}\}$, we have to join every found conflict from Γ with $\mathcal{H}(n)$ before adding them to the potential explanations Con (line 14).

Next, we can use a useful MXP property from Theorem 2.2.3. If Γ does not contain a conflict of size greater than 1, then there are no more conflicts in the given branch. Therefore, we expand the HS-tree under a given node only if Γ contains at least one conflict of size greater than 1 (line 15). Then, we label the node with $\text{Abd}(M) \setminus \mathcal{H}(n)$, where $M \in \text{Mod}$ is a model of $\mathcal{K} \cup \{\neg\mathcal{O}\} \cup \mathcal{H}(n)$ and create its successors (lines 16–17).

After completing the HS-tree T, we must filter out explanations that are not desired and return the resulting set of explanations $\mathcal{S}_\mathcal{E}$ (line 21). We have to check explanations at the end because modifications of `findConflict` inputs can cause non-minimal explanations to be found. We cannot determine that these explanations are non-minimal right away, only when we later find a corresponding explanation that is smaller.

Properties of the MHS-MXP algorithm: MHS-MXP is a *complete* algorithm that has all the advantages of the MHS algorithm. Thanks to the use of MXP, however, it also has other properties that give it an advantage over MHS.

Algorithm 2.3 MHS-MXP($\mathcal{K}$,O,Abd)

Input: knowledge base $\mathcal{K}$, observation O, set of abducibles Abd
Output: set $\mathcal{S}_\mathcal{E}$ of all explanations of $\mathcal{P} = (\mathcal{K}, O)$ of the class Abd

```
Con ← {}                                                    ▷ Set of conflicts
Mod ← {}                                                    ▷ Set of cached models
if ¬isConsistent(K ∪ {¬O}) then
    return "nothing to explain"
else if Abd(M) = ∅ where Mod = {M} then
    return S_E = ∅
end if
T ← (V = {r}, E = ∅, L = ∅)                                 ▷ Init. HS-Tree
while there is next node n in T w.r.t. BFS do
    if n can be pruned then
        prune n
    else
        ⟨_, Γ⟩ ← FindConflicts(K ∪ {¬O} ∪ H(n), Abd \ H(n))
        Con ← Con ∪ {H(n) ∪ γ | γ ∈ Γ}
        if ∃γ ∈ Γ : |γ| > 1 then                           ▷ Extend HS-tree under n
            L(n) ← Abd(M) \ H(n) for some M ∈ Mod s.t. M ⊨ H(n)
            for each σ ∈ L(n) create new σ-successor n_σ of n
        end if
    end if
end while
return S_E ← {γ ∈ Con | γ is desired}
function isConsistent(K)
    if there is M ⊨ K then
        Mod ← Mod ∪ {M}
        return true
    else
        return false
    end if
end function
```

From Observation 2.2.3 and Theorem 2.2.3, if all the explanations (including undesired) were of size 1, then MHS-MXP will find them all at the first `findConflict` call and the search may end. MHS would find the explanations at the first level of the HS-tree. Moreover, MHS cannot determine that it already has all the explanations, so it would have to continue building the HS-tree. We can generalize this property and say that after processing the HS-tree to depth n, all explanations up to size $n + 1$ are guaranteed to be found. In the case of MHS, at this point, we only have explanations up to size n.

The problem with MHS-MXP is that it finds all kinds of conflicts, not just desired explanations. If Abd contains such conflicts, for example $\{a\colon A, a\colon \neg A\}$, where $a \in N_I$ and $A \in N_C$, it will significantly slow down the search. In such cases, MHS-MXP loses its advantage over MHS, which does not have such a problem, and can potentially achieve worse results.

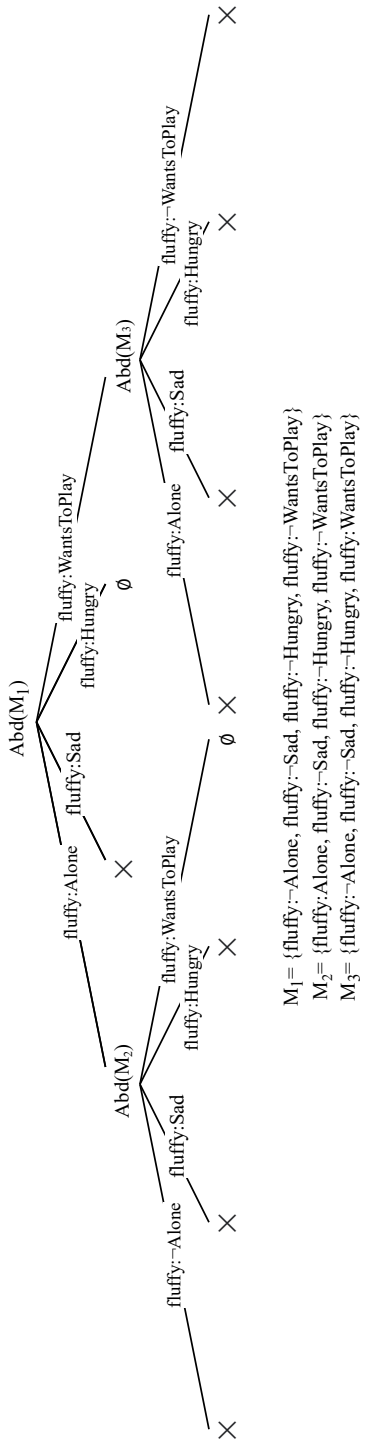


Figure 2.2: HS-tree for ABox abduction problem in Example 2.1.1

Chapter 3

Original implementation

In this chapter, we will look at the original version [4] of the experimental abduction solver implementation, which was our starting point in further development. Solver allows to use the MHS-MXP algorithm or the MHS algorithm to solve an ABox abduction problem. We will briefly describe points of original implementation that are relevant to this work. The source code of the original version is available on GitHub[1].

The abduction solver is implemented in Java. Dependency management is provided by Maven. Knowledge bases are represented in the form of ontology files. For working with ontologies the OWL API [12] interface is used. Reasoning over ontologies, such as consistency checking and model extraction, is left to the external OWL API reasoner, which is used as a black box. The original version of the solver supports work with 3 reasoners: *Hermit*, *JFact* and *Pellet*.

The solver is still under development. There is plenty of room for changes, corrections and improvements in the implementation.

[1] `https://github.com/IvetBx/MHS-MXP-algorithm`

3.1 Running the solver

In order for the solver to be able to solve the abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$, we must first provide it with $\mathcal{K}$ and $\mathcal{O}$, for which the solution should be found. The solver receives this information in the form of a structured input file.

The solver application can be run through the command line by entering a command of the form: `java -Xmx4096m -jar <solver JAR file> <path to an input file>`, where `-Xmx4096m` allows allocating more memory, which is highly recommended.

Another way to run the solver is directly through the `src/main/java/-Main.java` class in a Java IDE.

As an output, the solver creates text log files that contain the found explanations and time data. We will provide more information about the structure of the input file and output log files in the following sections.

3.2 Inputs

The abduction solver receives a structured input file as a parameter. The input file contains one switch per line. Mandatory switches are **`-f`** and **`-o`**, other switches are optional. We will describe all the switches that can be set in the input file.

`-f: <string>` a relative path to the ontology file, which represents the knowledge base $\mathcal{K}$.

`-o: <ontology>` observation $\mathcal{O}$ in the form of an ontology (in any ontology syntax), which has to be written in one line.

`-t: <positive integer>` the time after which the search for explanations terminates. By default, it is not set.

`-d: <positive integer>` the depth of the HS-tree, when the search terminates. For example, for `-d: 2` search terminates after completing

level 1 of HS-tree. By default, it is not set.

`-r: "hermit"|"jfact"|"pellet"` the name of the reasoner to be used. The default reasoner is Hermit.

`-mhs: <boolean>` using the MHS algorithm for explanations search. Set to `false`, by default.

`-n: <boolean>` allowing negated assertions in explanations. Set to `true`, by default.

`-l: <boolean>` allows assertions of form $i, i\colon R$ in explanations, i.e. individual i can be in role R with itself (it is also called *looping*)

Abducibles

Defining abducibles using entities they can contain:

`-aI: <IRI of an individual>` defines individual that is used in abducibles. We can define more abducible individuals by writing multiple `-aI` switches. By default, all invididuals from $\mathcal{K}$ and $\mathcal{O}$ are used.

`-aC: <IRI of a class>` defines class that is used in abducibles. We can define more classes by writing multiple `-aC` switches. By default, all classes from $\mathcal{K}$ and $\mathcal{O}$ are used.

`-aR: <IRI of an object property>` defines object property that is used in abducibles. We can define more object properties by writing multiple `-aR` switches. By default, all object properties from $\mathcal{K}$ and $\mathcal{O}$ are used.

There is also another way by which it is possible to define a list of individuals in one `-aI` switch. The list is wrapped in curly braces and each individual is in one line.

```
-aI: {
<IRI of an individual 1>
<IRI of an individual 2>
}
```

Same can be done with `-aC` (defining a list of classes) and `-aR` (defining a list of object properties).

Defining abducibles directly by enumerating assertions:

`-abd: <ontology>` a complete ontology (in any ontology syntax), which has to be written in one line.

Another way is to list only assertions. In this case, Manchester Syntax[13] needs to be used.

```
-abd: {
<assertion 1>
<assertion 2>
}
```

`-abdF: <string>` a relative path to the ontology file, assertions from the ontology will be used as abducibles.

Abducibles cannot be defined by combining different definition types. Either they are defined using entities (`-aI`, `-aC`, `-aR`), using an ontology or list of assertions (`-abd`), or from the ontology file (`-abdF`).

3.3 Outputs

As an output for a given input, the solver produces several log files. Each file contains explanations and times, but groups them in different ways. Time in the logs is given in seconds. Logs can be divided into 2 basic types: *final* and *partial*.

Final logs

Final logs are created only after the search for explanations is complete (it either ended normally and all explanations have been found or was interrupted by the depth limit or timeout). The found explanations are filtered and only the desired ones are written in the logs.

Hybrid log

`<time>__<input file name>__hybrid.log`
Hybrid log is a final log which contains desired explanations of a certain size in each line (except the last, which contains the total running time). Line with explanations has the form: `<size n>;<explanation count>; <level completion time>;{<explanations of size n>}`.

Level log

`<time>__<input file name>__hybrid_level.log`
Level log is a final log which contains desired explanations founded in a certain level in each line (except the last, which contains the total running time). Line with explanations has the form: `<level l>;<explanation count>; <level l completion time>;{<explanations found in level l>}`.

Explanation times log

`<time>__<input file name>__hybrid_explanation_times.log`
Explanation times log is a final log which contains desired explanations and time when they were found. Line has the form: `<time t>;<explanation found in the time t>`

Partial logs

Partial logs are created while solving the abduction problem is running. They help us to have an overview of the progress along the run. They record, for example, possible explanations which were found after passing one level.

However, these explanations are only possible explanations and therefore may not be desired.

Partial explanations log

`<time>__<input file name>__hybrid_partial_explanations.log`
Partial explanations log is a partial log with the same structure as **hybrid log** (explanations are grouped by size).

Partial level explanations log

`<time>__<input file name>__hybrid_partial_level_explanations.log`
Partial level explanations log is a partial log with the same structure as **level log** (explanations are grouped by the level they were found in).

MHS logs

When solving the abduction problem using MHS-MXP, all the mentioned logs are produced. When using the MHS algorithm, however, only some are produced: **hybrid log**, **explanation times log** and **partial explanations log**. The reason for this is that other logs would be redundant. For the MHS algorithm, the grouping explanations according to their size is identical to grouping them according to their levels.

Location of logs

Logs of inputs that used the MHS algorithm have a different location than the logs of inputs that used the MHS-MXP algorithm.

- location of the MHS-MXP logs:
  `logs/<reasoner name>/<input ontology path>`
- location of the MHS logs:
  `logs_mhs``/<reasoner name>/<input ontology path>`

3.4 Implementation structure

In this section, we will provide a brief description of the solver source code structure. The original implementation source code consists of 5 main and 3 auxiliary components.

Main components

Algorithms: the main component used to execute the MHS-MXP (or MHS) algorithm to find all solutions for a given input.

Parser: used for reading and processing the input.

File logger: used for creating log files and writing records to them.

Reasoner: used to initialize and work with a reasoner.

Timer: used to measure runtime.

Auxiliary components

Common: contains auxiliary things, used throughout the project, for example, a configuration and easier axiom display.

Models: contains classes representing structures used in the algorithm, such as explanation, abducibles and observation.

Application: used to exit the application, for example, in the case of invalid input.

3.5 Model extraction

The main problem of the original implementation was model extraction which did not work correctly in all cases. We discovered this problem during the testing of models, which was carried out because we did not get all the explanations for some inputs. Problematic cases turned out to be cases

where the KB whose model we wanted to obtain contained a concept union assertion.

Obtaining a valid model of the currently loaded KB $\mathcal{K}_C$ (Algorithm 3.1) is done in such a way that we gradually go through the individuals contained in abducibles. For each individual i, we find out which atomic concepts i should belong to (`NewTypes`) in order to form a valid model of $\mathcal{K}_C$. We assign i to these atomic concepts and add created assertions to `Model` that is our representation of the model. Then, for all other atomic concepts (`NewNotTypes`), we assign i to their complement and add these assertions to `Model`.

There is also an optimization, which we will refer to as *model trimming*. We remove from `Model` such assertions which are directly contained in $\mathcal{K}'$ (`KnownTypes` and `KnownNotTypes`). Those assertions could not be in an explanation, because they would automatically make the original knowledge base $\mathcal{K}$ inconsistent with such an explanation (note that explanations are built from *negated* assertions from models).

Algorithm 3.1 Original model extraction

Input: Original knowledge base with negated observation $\mathcal{K}'$, abducibles Abd
Output: Model of currently loaded knowledge base $\mathcal{K}_C$

Model $\leftarrow$ {}
for individual i in Abd **do**
    AssignTypesToIndividual(i)
end for
return Model

function AssignTypesToIndividual(i)
    KnownTypes $\leftarrow$ all concepts C from $\mathcal{K}'$ such that $i\colon C$
    KnownNotTypes $\leftarrow$ all concepts C from $\mathcal{K}'$ such that $i\colon \neg C$
    NewTypes $\leftarrow$ GetTypes(i, false) $\setminus$ KnownTypes
    NewNotTypes $\leftarrow$ all $A \in N_C \setminus$ KnownNotTypes $\setminus$ KnownTypes $\setminus$ NewTypes
    Model $\leftarrow$ Model $\cup \{i\colon A \mid A \in \text{NewTypes}\} \cup \{i\colon \neg A \mid A \in \text{NewNotTypes}\}$
end function

The main problem lies in the fact that getting the atomic concepts to which the individual i should belong to in a model of $\mathcal{K}_C$ is done using the method `getTypes(`i`, false)` from OWL API. This method solves the entailment and therefore returns all atomic concepts to which the given individual

must belong in each model of $\mathcal{K}_C$. That is a problem if this individual is assigned to the concept union (Example 3.5.1).

Example 3.5.1 (Problematic case) *Let us have an individual i and atomic concepts A_1 and A_2. The loaded knowledge base contains assertion i: $A_1 \sqcup A_2$.*

Then when we call method `getTypes(i, false)`*, we get an empty set, because individual i does not clearly belong to any of the concepts A_1 or A_2.*

However, this is a problem, because if we want to get a model of the given knowledge base, the individual i must belong to at least one of them.

Correct model extraction is very important for the correct functioning of both the MHS and MHS-MXP algorithms. It is needed for the proper creation of the HS-tree, which ensures that we are able to find all the explanations. Therefore, one of the main goals is to correct this problem.

Part II

Our contribution

Chapter 4

Model extraction

As we already mentioned, proper model extraction is crucial for the correct functioning of both the MHS-MXP algorithm and the MHS algorithm. Finding a way to correctly extract a model is, however, a non-trivial task because model extraction is not one of the standard DL reasoners features.

In this chapter, we will look at possible solutions we explored to solve the problem of model extraction.

4.1 OWLKnowledgeExplorerReasoner

`OWLKnowledgeExplorerReasoner`[1] is an interface that provides access to the completion tree (or completion graph), which we denoted CTree for short. In Section 1.4, we described what it is, showed how such a CTree looks and how it is created. Therefore, if we have access to the CTree, we should be able to read from it the label of an individual, which contains concepts it belongs to. It should also be possible to obtain roles to which a pair of individuals belong from their edge label. As we showed in Example 1.4.1, this should allow us to correctly extract a model of a knowledge base. Moreover, including roles in a model enables expanding our implementation so that it can provide explanations with role assertions.

[1] `https://owlcs.github.io/owlapi/apidocs_4/org/semanticweb/owlapi/reasoner/knowledgeexploration/OWLKnowledgeExplorerReasoner.html`

The label of an individual i can be obtained in the following way. For the individual i, we first get the vertex v that represents it in the CTree using the `getRoot(`e`)` method, which receives the individual i in the form of a nominal $e = \{i\}$. Next, we call the `getObjectLabel(`v`,false)` method with the obtained vertex, from which we obtain the atomic concepts to which i belongs.

Finding the roles to which a pair of individuals belong is a slightly more complicated process. So let us have an individual i. We want to obtain all the roles to which i belongs *as a subject*, i.e. such roles R for which holds $i^{\mathcal{I}} \in \exists R.\top^{\mathcal{I}}$, where $\mathcal{I}$ is a model of the ontology currently loaded in the reasoner. As in the case with the label, we first obtain a vertex v that represents individual i using the method `getRoot(`e`)`, where $e = \{i\}$. Then all the roles to which individual i can belong are obtained using the method `getObjectNeighbours(`v`, false)`. Next, we also have to find what individuals are *objects* in an obtained role R with the individual i. So we are searching for such $y : (i^{\mathcal{I}}, y^{\mathcal{I}}) \in R^{\mathcal{I}}$, where $\mathcal{I}$ is a model of the ontology currently loaded in the reasoner. The method of the same name `getObjectNeighbours(`v`, `R`)` is used for this. The method returns a collection of vertices that represent individuals y which are in the role R with the individual i.

`OWLKnowledgeExplorerReasoner` interface is part of the OWL API [12]. It extends the classic `OWLReasoner`. However, not all reasoners that can be accessed via the OWL API implement this interface. Of the reasoners supported by our implementation, only the JFact reasoner implements it.

While trying this interface, we encountered certain problems and bugs, from which it was not clear whether it provides the functionality we require.

4.2 DIG Interface

Since some problems appeared during the `OWLKnowledgeExplorerReasoner` research, we decided to explore other possible solutions. We searched for

other Java interfaces that enable work with ontologies and provide reasoning over them, and we found *The DIG Description Logic Interface (DIG Interface)* [5, 6].

DIG Interface provides common access to DL reasoners. The interface is lightweight and offers only a minimal set of reasoning tasks. Among those provided tasks are, for example, subsumption checking.

Communication with a reasoner is held via HTTP requests, which consist of 3 types: management, tell and ask.

- *Management requests* correspond to loading and releasing a knowledge base, and also acquiring information about the used DL reasoner, like its version and name.
- *Tell requests* enable us to modify loaded knowledge base (e.g. define new concept name or add concept assertion).
- *Ask requests* consists of queries about the loaded knowledge base (e.g. query if concept C subsumes concept D, query if an individual i is an instance of concept C).

DL expressions, requests and answers are encoded in XML format.

However, DIG Interface is not suitable for our needs and did not deliver the functionality that we require. There does not appear to be a direct way to test the overall consistency of a knowledge base through the DIG Interface, which is the main functionality we need from a reasoner. Model extraction is not provided by the DIG interface either, as it is a non-standard functionality and the interface is focused only on basic tasks.

4.3 JFact Reasoner

The last option we explored was to use one reasoner directly in our solver and not access it through the OWL API. This approach will ensure that we will have direct access to the reasoner objects and will be able to extract any

necessary information from it. However, the disadvantage of this approach is that we cannot consider the used reasoner as a true black box anymore. We chose to explore the JFact reasoner.

JFact is a DL reasoner, which is a tool that enables us to make reasoning tasks over a knowledge base, like consistency check which is needed in our abduction solver. JFact is tableau-based which means it uses the tableau algorithm (described in Section 1.4) to check the consistency of a knowledge base. Therefore, we know that during the consistency check, a CTree is created, from which we would be able to obtain a model of a knowledge base.

JFact is implemented in Java, does not have command-line support, and is designed to be used purely through the OWL API. So, it would require some modifications to adjust it for direct access. We downloaded the source code from the JFact GitHub repository[2], examined it and did experiments on it to obtain a label for individuals. We did not find any other documentation or materials for the JFact reasoner, so we only gathered information by searching the source code, which was sometimes briefly described, and by conducting small experiments with it.

A CTree T is represented by the `DlCompletionGraph` class. A vertex of T is represented by the `DlCompletionTree` class. Vertices that represent individuals can be obtained using the `isNominalNode()` method. We can also obtain the label of a vertex by calling the method `label()` on a given vertex. The vertex label can look like this, for example: `[8, 10, -6, -4, -9, 5, -7]`, where the given numbers represent concepts. The problem arises when we want to find out which concept a number represents. In the case of reasoning itself, there is no need for a reverse translation. This functionality of reverse translation is provided by `KnowledgeExplorer` class, which is also used in the JFact implementation of the `OWLKnowledgeExplorerReasoner` interface. So at this point, we started to closely examine the implementation of the given interface and uncover problems in it, for which it could not be

[2] `https://github.com/owlcs/jfact`

used. We programmed the translation of the vertex label into a readable form and found out that during the translation there is definitely an error which causes the atomic concepts to be lost from the original label. We will show this issue in a concrete example 4.3.1.

Example 4.3.1 (Label translation with missing concepts) *Let us say, we obtained the label:* `[8, 10, -6, -4, -9, 5, -7]` *from a vertex of a completion tree which was created after consistency check.*

We get the translation which looks like this:

$\{\mathsf{jack}\} \leftarrow 8$

$\neg\mathsf{Tired} \sqcap \neg\mathsf{Sad} \sqcap \neg(\neg\mathsf{Tired} \sqcap \neg\mathsf{Stressed}) \leftarrow 10$

$\neg\mathsf{Tired} \leftarrow -6$

$\neg\mathsf{Sad} \leftarrow -4$

$\neg(\neg\mathsf{Tired} \sqcap \neg\mathsf{Stressed}) \leftarrow -9$

$\neg(\neg\mathsf{Sad} \sqcap \mathsf{Tired}) \leftarrow -7$

We can see that the translation of element 5, which should be translated as Stressed*, is missing.*

After further research, we discovered the exact place where this issue occurs and fixed it. This fix ensured that we could return to the original approach via the OWL API. For model extraction, we can after all use the `OWLKnowledgeExplorerReasoner` interface, which will use our experimental version of JFact reasoner, where this main problem is fixed. So we can go back to the true black box approach.

Chapter 5

Implementation

In this chapter, we describe our contribution to the implementation of the abduction solver. The most important part, which was necessary for correct working of our solver, is a new version of the model extraction. The problem of the previous version was discussed in detail in Section 3.5. In addition, we have made other modifications such as expanding explanations so they can include role assertions, defining types of relevance for multiple observation, log files adjustment and other improvements to our solver.

5.1 Model extraction

To extract a model, we chose to use the `OwlKnowledgeExplorerReasoner` interface[1]. As we have already described in Section 4.1, this interface is implemented by only one of the reasoners that are supported by our abduction solver, which is the JFact reasoner. For this reason, our solver can only work with this reasoner for now and it is not possible to choose another in the input file.

However, there were some complications. JFact implementation of this interface contained some bugs and therefore we had to solve these problems first to make it work.

[1] `https://owlcs.github.io/owlapi/apidocs_4/org/semanticweb/owlapi/reasoner/knowledgeexploration/OWLKnowledgeExplorerReasoner.html`

5.1.1 Fixing JFact implementation of the interface

Initial difficulties were solved by Ignazio Palmisano, one of the JFact reasoner developers, after we contacted him[2]. The problem was that the method for getting the label of an individual `getObjectLabel` ended with an error on any input. The reason for that was that the label also contained complex concepts and JFact tried to return them but failed because the translation of a complex concept to OWL API representation of a concept expression `OwlClassExpression` was not implemented. The problem was solved so that complex concepts in a label are ignored. To extract a model we mainly need to obtain atomic concepts, so this fix satisfactorily solves the problem.

During further research, we came across other relatively serious bugs in the `OwlKnowledgeExplorerReasoner` implementation. There was another bug in method `getObjectLabel`, which caused that method was not able to return atomic concepts, so it always returned an empty set. This problem was described in more detail in Section 4.3. Another problem arose when we tried to incorporate role assertions in our models. The method `getObjectNeighbours` ended with an error for any ontology that contained role restrictions like symmetry, reflexivity and transitivity.

We were not able to get in contact with the JFact developers again, but we notified them about the problems that we found. Then, we created our experimental version of JFact[3] with potential fixes for those bugs and used this version in our solver.

Later, during the evaluation, we encountered two other difficulties. However, further correction of JFact implementation problems was beyond the scope of our work. After all, we only take a reasoner as a ready-made package whose functionality we use. Therefore we found another way to deal with them.

The first problem occurred with method `getObjectLabel`, which did not

[2]https://stackoverflow.com/questions/71818598/owlknowledgeexplorerreasoner-getobjectlabel-always-ends-in-error-unreachable-s/71898063?noredirect=1#comment127079531_71898063

[3]`https://github.com/boborova3/jfact/tree/test4`

work correctly for ontologies with data properties in them. In the correct case, they should be ignored, but the method ended with an error when it encountered them. We were able to deal with this error quickly and we removed the data properties from the ontologies used in the evaluation. Data properties are ignored and have no influence on MHS or MHS-MXP algorithms, so we could afford this change.

Another problem appeared, using the method `getObjectNeighbors`. The method ended with an error on some ontologies from the evaluation. We did not manage to find the exact problem, but it is possible that the problem is caused by other role restrictions, such as domain and range. In the evaluation, we resolved this issue by choosing only a group of inputs that is not problematic, so we did not allow role assertions in explanations.

It is possible that the JFact implementation of the `OwlKnowledgeExplorerReasoner` interface contains other hidden flaws. For evaluations with other inputs, it would be necessary to pay special attention to those shortcomings and further examine the JFact reasoner.

5.1.2 Obtaining the concept assertions of a model

Obtaining the concept assertions to form our model representation takes place in the same way as in the original implementation. The only change is that to find out which atomic concepts an individual should belong to in order to form a valid model of the currently loaded KB $\mathcal{K}_C$, we use the `getObjectLabel` method from the `OwlKnowledgeExplorerReasoner` interface instead of the `getTypes` method that was used previously (Algorithm 3.1). We described how method `getObjectLabel` works in Section 4.1.

The new version is shown in Algorithm 5.1. Obtaining concept assertions for a given individual is contained in the `AssignTypesToIndividual` method.

Algorithm 5.1 Model extraction

Input: Original knowledge base with negated observation $\mathcal{K}' = (\mathcal{T}', \mathcal{A}')$, abducibles Abd
Output: Model of currently loaded knowledge base $\mathcal{K}_C$

Model $\leftarrow \{\}$

for individual i in Abd **do**
    AssignTypesToIndividual(i)
    AssignRolesToIndividual(i)
end for
return Model

function AssignTypesToIndividual(i)
    KnownTypes $\leftarrow$ all concepts C from $\mathcal{K}'$ such that $i\colon C$
    KnownNotTypes $\leftarrow$ all concepts C from $\mathcal{K}'$ such that $i\colon \neg C$

    $e \leftarrow$ nominal in the form of $\{i\}$
    $v \leftarrow$ GetRoot(e)
    NewTypes $\leftarrow$ GetObjectLabel(v, false) $\setminus$ KnownTypes
    NewNotTypes $\leftarrow$ all $A \in N_C \setminus$ KnownNotTypes $\setminus$ KnownTypes $\setminus$ NewTypes

    Model $\leftarrow$ Model $\cup \{i\colon A \mid A \in$ NewTypes$\} \cup \{i\colon \neg A \mid A \in$ NewNotTypes$\}$
end function

function AssignRolesToIndividual(i)
    KnownRoleAssertions $\leftarrow$ all role assertions $i\colon R$ from $\mathcal{K}'$, where $R \in N_R$
    KnownNotRoleAssertions $\leftarrow$ all role assertions $i\colon R$, where $R \in N_R$ and $i\colon \neg R \in \mathcal{A}'$

    NewRoleAssertions $\leftarrow \{\}$
    Roles $\leftarrow$ GetObjectNeighbours(v, false)
    for role R in Roles **do**
        for vertex v in GetObjectNeighbours(v, R) **do**
            $y \leftarrow$ individual corresponding to vertex v
            NewRoleAssertions $\leftarrow$ NewRoleAssertions $\cup \{i, y\colon R\}$
        end for
    end for

    NewRoleAssertions $\leftarrow$ NewRoleAssertions $\setminus$ KnownRoleAssertions
    NewNotRoleAssertions $\leftarrow \{i, x : R \mid x \in N_I, R \in N_R\} \setminus$ KnownNotRoleAssertions $\setminus$ KnownRoleAssertions $\setminus$ NewRoleAssertions

    Model $\leftarrow$ Model $\cup \{$assertion $\mid$ assertion $\in$ NewRoleAssertions$\}$
    Model $\leftarrow$ Model $\cup \{$negated assertion $\mid$ assertion $\in$ NewNotRoleAssertions$\}$
end function

5.1.3 Obtaining the role assertions of a model

Since we are already able to obtain correct models formed by concept assertions, we can take the next step and expand them with the role assertions. Adding role assertions to models will allow the solver to provide explanations that contain role assertions. In other words, the abduction solver will be able to explain the given observation not only with concept assertions but also with role assertions.

The new version of model extraction with role assertions is shown in Algorithm 5.1. Obtaining role assertions in order to create a model, is done similarly to obtaining concept assertions. We go through the individuals contained in the abducibles and for each individual i, we get all the role assertions where i is assigned as a subject, i.e. $\{i, x\colon R \mid \mathcal{I} \models \mathcal{K}_C, (i,x)^{\mathcal{I}} \in R^{\mathcal{I}}, x \in N_C, R \in N_R\}$, and all *negated* role assertions where i is assigned as a subject, i.e. $\{i, y\colon \neg S \mid \mathcal{I} \models \mathcal{K}_C, (i,y)^{\mathcal{I}} \notin S^{\mathcal{I}}, y \in N_C, S \in N_R\}$. Obtaining role assertions for individual i is contained in the `AssignRolesToIndividual` method.

To obtain role assertions where individual i is assigned as a subject, we use methods `getObjectNeighbors` from the `OwlKnowledgeExplorerReasoner` interface. We described these methods in more detail in Section 4.1. *Negated* role assertions where individual i is assigned as a subject are then all other possible role assertions with i as a subject, where i was not assigned in the previous step.

When obtaining role assertions for an individual, we also use similar optimizations and do not add to the model assertions that are directly contained in $\mathcal{K}'$ (`KnownRoleAssertions` and `KnownNotRoleAssertions`).

5.2 Relevance for multiple observations

As we mentioned in Chapter 2, when solving the problem of ABox abduction, we are only interested in a certain group of explanations, which we call *desired*. One of the conditions for an explanation to be desired is for it

to be relevant (Definition 5.2.1).

Definition 5.2.1 (Relevant explanation) *An explanation $\mathcal{E}$ of ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ is relevant if $\mathcal{E} \not\models \mathcal{O}$.*

Definition 5.2.1 is clear in the case of single observations. However, we can think about how to define the relevance of the explanation in the case of multiple observation, which is an observation consisting of several assertions $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, ..., \mathcal{O}_n\}$. There are basically 2 ways:

- We can require the explanation to be relevant with respect to each assertion from the observation. We defined this as **strict relevance** (Definition 5.2.2).

  Definition 5.2.2 (Strictly relevant explanation) *An explanation $\mathcal{E}$ of ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ is strictly relevant if for every $\mathcal{O}_i \in \mathcal{O}$ holds that $\mathcal{E} \not\models \mathcal{O}_i$.*

- We can require the explanation to be relevant with respect to at least one assertion from the observation. We defined this as **partial relevance** (Definition 5.2.3).

  Definition 5.2.3 (Partially relevant explanation) *An explanation $\mathcal{E}$ of ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ is partially relevant if there exists $\mathcal{O}_i \in \mathcal{O}$ such that $\mathcal{E} \not\models \mathcal{O}_i$.*

Different types of explanation relevance in the case of multiple observation can be seen in Example 5.2.1

Example 5.2.1 (Different types of explanation relevance) *Let us have some knowledge base $\mathcal{K}$ and observation $\mathcal{O} = \{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Mother}\}$. We solved the ABox abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ and got the set of possible explanations $\mathcal{S}$ such that:*

$$\mathcal{S} = \{\{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Mother}\},$$

$\{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Mother}\}$,

$\{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}$,

$\{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}\}$

Then we can determine what type of relevance individual explanations have. For explanation $\mathcal{E}_1 = \{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Mother}\}$:

$\mathcal{E}_1 \models \mathsf{jack}\colon \mathsf{Father}$ *and* $\mathcal{E}_1 \models \mathsf{amy}\colon \mathsf{Mother}$

so $\{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Mother}\}$ *is an irrelevant explanation.*

For explanation $\mathcal{E}_2 = \{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Mother}\}$:

$\mathcal{E}_2 \not\models \mathsf{jack}\colon \mathsf{Father}$ *and* $\mathcal{E}_2 \models \mathsf{amy}\colon \mathsf{Mother}$

so $\{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Mother}\}$ *is a partially relevant explanation.*

For explanation $\mathcal{E}_3 = \{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}$:

$\mathcal{E}_3 \models \mathsf{jack}\colon \mathsf{Father}$ *and* $\mathcal{E}_3 \not\models \mathsf{amy}\colon \mathsf{Mother}$

so $\{\mathsf{jack}\colon \mathsf{Father}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}$ *is a partially relevant explanation.*

For explanation $\mathcal{E}_4 = \{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}$:

$\mathcal{E}_4 \not\models \mathsf{jack}\colon \mathsf{Father}$ *and* $\mathcal{E}_4 \not\models \mathsf{amy}\colon \mathsf{Mother}$

so $\{\mathsf{jack}\colon \mathsf{Parent}, \mathsf{jack}\colon \mathsf{Male}, \mathsf{amy}\colon \mathsf{Parent}, \mathsf{amy}\colon \mathsf{Female}\}$ *is a strictly relevant explanation.*

There is no right answer as to which of these definitions is correct for multiple observations. Therefore, we have enabled both. To choose the type of relevance, a line in the form: `-sR: <boolean>` is inserted into the input file. If the `<boolean>` equals `true`, strict relevance is used, if it equals `false`, then partial relevance is used. The default type of relevance is set to strict relevance.

5.3 Log files adjustment

Our solver stores outputs in the form of log files. We described the types of log files in more detail in Section 3.3. Our goal was to modify these logs to make them easier to work with. First, we focused on making their location more appropriate and then added new types of logs that could be useful.

5.3.1 Location of logging files

All logs are located inside a log folder, which is in the same directory as the executive JAR file of the abduction solver. Inside the log folder itself, however, there is a relatively intricate hierarchy of folders before we get to the log files themselves. Original relative path to log files of a given input:
`./<log folder>/<reasoner name>/<input ontology path>`

As we can see, in the previous version, the hierarchy contains the name of the reasoner, which was used for the given input. This information is not needed in the current version, since the solver supports only the JFact reasoner. If it is necessary to record this information in the future, it may be recorded in a log file.

Another thing that unnecessarily deepened the folder hierarchy was that the whole relative path of the ontology from the input file is used.

We also wanted to divide the log files in a meaningful way, for example, each input would have its own folder with its log files from different runs.

Based on these observations, we proposed a new default relative path to log files of a given input:
`./<logs folder name>/<input ontology name>/<input file name>`

However, since this decision may not suit each user or situation, we decided to add the possibility of entering a desired relative log path into the input file. The custom path can be defined by adding a line of the form `-output: <relative path>` to a input file.

5.3.2 New types of log files

When proposing a new location for log files, we also thought about other things that we could improve and what information we were missing when working with the solver. We have therefore created 2 new types of log files:

Information log: `<time>__<input file name>__info.log`

Log contains basic information about how the input was set in a given run, for example, timeout, maximum depth or enabling negation in explanations. Later, we also added messages for a user.

Error log: `<time>__<input file name>__error.log`

Log records an error that could have occurred during the construction of the HS-tree.

5.4 Bug fixes and improvements

While working with the solver, we managed to discover some shortcomings, which we subsequently corrected. These were mostly bugs or things that needed a small adjustment. In this section, we will describe the two most significant of them, which had a major impact on the correctness and functioning of the solver.

5.4.1 Interpretation of the consistency check result

The first discovered problem was an incorrect consistency check during HS-tree building. Consistency was inferred based on whether the found model object was empty. However, since the solver contains *model trimming* optimization which removes assertions from a model (mentioned in Section 3.5), a model could be empty even in situations where the tested knowledge base was consistent. Improper checking in some cases caused the program to infinitely loop while searching for conflicts (Example 5.4.1).

Example 5.4.1 (Incorrect interpretation of the consistency check) *Let us have a given vocabulary:* $N_I = \{a\}$, $N_C = \{A, B\}$ *and* $N_R = \{\}$. *We also have a knowledge base* $\mathcal{K} = \{a\colon A\}$ *and an observation* $\mathcal{O} = \{a\colon B\}$. *Then* $\mathcal{K}' = \mathcal{K} \cup \{\neg\mathcal{O}\} = \{a\colon A, a\colon \neg B\}$.

Let us check the consistency of $\mathcal{K}'$. *We can see that* $\mathcal{K}'$ *is consistent and we get the model* $\mathcal{M} = \{a\colon A, a\colon \neg B\}$. *However,* $\mathcal{M}$ *is empty after applying model trimming that removes assertions contained in* $\mathcal{K}'$ *from* $\mathcal{M}$. *Based on this, the original version of the implementation infers that* $\mathcal{K}'$ *is inconsistent, although it is not true.*

It was necessary to change this condition and decide on the consistency of a given knowledge base based on another parameter. Therefore, we saved the result of the consistency check in a separate boolean variable `modelIsValid`, and we ask the model that was created from the consistency check if it is valid, which will return the correct answer.

5.4.2 Consistent explanations check

As we mentioned, when solving the abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$ we are only interested in explanations with certain properties. One of these properties is that the explanation $\mathcal{E}$ should be consistent, i.e. $\mathcal{K} \cup \mathcal{E}$ is consistent. This check was not properly implemented in the previous version. Instead of directly checking the consistency of the $\mathcal{K} \cup \mathcal{E}$, the original ontology $\mathcal{K}$ was constructed by removing the negated observation from the ontology with negated observation $\mathcal{K}' = \mathcal{K} \cup \{\neg\mathcal{O}\}$. This could lead to incorrect construction of the original knowledge base and receiving inconsistent explanations (Example 5.4.2).

Example 5.4.2 (Inconsistent explanation passes consistency check) *Let us have a given vocabulary:* $N_I = \{a\}$, $N_C = \{A, B\}$ *and* $N_R = \{\}$. *We also have a knowledge base* $\mathcal{K} = \{A \sqsubseteq B, a\colon A\}$ *and an observation* $\mathcal{O} = \{a\colon \neg A\}$. *Then* $\mathcal{K}' = \mathcal{K} \cup \{\neg\mathcal{O}\} = \{A \sqsubseteq B, a\colon A\}$.

After solving the abduction problem $\mathcal{P} = (\mathcal{K}, \mathcal{O})$, we obtained the only solution, the explanation $\mathcal{E} = \{a\colon \neg B\}$.

Let us check if the explanation is consistent according to the original implementation. First, we construct the original knowledge base $\mathcal{K}$ by removing the negated observation $\{\neg\mathcal{O}\}$ from $\mathcal{K}'$. We get $\mathcal{K}_{ver2} = \{A \sqsubseteq B\}$. We can see that $\mathcal{K} \neq \mathcal{K}_{ver2}$. From the consistency check of $\mathcal{K}_{ver2} \cup \mathcal{E}$, we find that it is consistent and $\mathcal{E}$ is therefore a consistent explanation.

Now, let us check the consistency of $\mathcal{K} \cup \mathcal{E}$ with the real original knowledge base $\mathcal{K}$. From $\mathcal{K}$, $a\colon A$ and $A \sqsubseteq B$. Then it must also be true that $a\colon B$. However, from $\mathcal{E}$, it must be true that $a\colon \neg B$. Individual a cannot be assigned to the concept B and to its complement $\neg B$ at the same time. There is a conflict, so $\mathcal{K} \cup \mathcal{E}$ is inconsistent. Therefore $\mathcal{E}$ is not a consistent explanation.

We fixed this check and now it is performed with the actual original knowledge base.

Chapter 6

Evaluation

In this chapter, we will describe two experiments we conducted to test the MHS-MXP algorithm. In the first experiment, we will examine how the size and the number of explanations affect the algorithm runtime. In the second experiment, we will look at what results the algorithm achieves on a group of various real-world ontologies. We will compare the MHS-MXP algorithm with the classic MHS algorithm.

In the previous work [4], various optimizations of the MHS-MXP algorithm were developed and tested. However, since none of them brought a significant improvement, we will use the classic version, which contains only Reiter's pruning optimizations.

We created one JAR file of our implementation, which includes both algorithms, MHS and MHS-MXP. We can specify which algorithm will be used in the input file. Since they are contained in the same implementation, their comparison is more accurate and there are no implementation differences.

However, we must note that this is not a plain version of the MHS algorithm, because it uses a few optimizations (which are also used in MHS-MXP), such as model trimming, which can reduce the size of the HS-tree that needs to be constructed.

The experiments were run on a device with the operating system Ubuntu 20.04 and Oracle Java SE Runtime Environment v1.8.0_201, inside a virtual

machine with 32BG RAM, using 8 cores (16 threads) of an Intel Xeon CPU E5-2695 v4 processor with 2.10GHz frequency. To measure the execution times in Java, we utilized the `ThreadMXBean` from the `java.lang.management` package. We measured user time, which excludes any system overhead and reflects the actual time consumed by the task. The maximum Java heap size was set to 4GB.

6.1 Experiment 1

In the first experiment, we decided to repeat the part of the previous evaluation [4] where the focus is on comparing the MHS-MXP algorithm and the standard MHS algorithm. We also want to observe the effect of the size and count of explanations on the MHS-MXP algorithm runtime.

We no longer have to modify the used ontology and create auxiliary concepts and axioms in it. This kind of approach was necessary for the previous version due to the limitations caused by the restrictive acquisition of models (described in Section 3.5). Since this issue has been fixed, we can make a proper comparison with different types of observations in inputs.

Unlike the previous evaluation, we will also include inputs with negated assertions in the explanations.

6.1.1 Methodology

Ontology

In this experiment, we used the LUBM (Lehigh University Benchmark [9]) ontology for all our test cases. The application domain of this ontology is the university and it contains concepts such as Student, Institute, Employee, Publication and ResearchWork. The LUBM ontology is considered a standard benchmark for testing various reasoning capabilities of OWL knowledge base systems. The basic metrics of this ontology are shown in Table 6.1.

concepts	roles	individuals	logical axioms
43	25	0	93

Table 6.1: The basic metrics of LUBM ontology.

Observations generating

In the experiment, we used the inputs generated in the previous work [4]. The goal was to generate observations so that we get explanations of different sizes for different test cases.

The LUBM ontology has a suitable size in terms of the concept count. However, it does not have a sufficiently complex structure, which is reflected in the size of the explanations. As a result, single observations in the form of an atomic concept assertion return explanations with a size of 1 at most. Since we want to obtain explanations of different sizes, we use complex observations, which were created as follows:

$$a\colon A_1 \sqcap \ldots \sqcap A_n$$

where $A_1, \ldots, A_n$ are atomic concepts from the ontology and a is a new individual. Generated observations contain only atomic concept assertions in their explanations.

An observation constructed in this way will have the largest explanation with a size of at most n. If $A_1, \ldots, A_n$ are independent and have no subconcepts, there will be no explanation (there would be only one possible explanation $\{A_1, \ldots, A_n\}$ which is not relevant). In case they are independent and have subconcepts, there will be at least 1 explanation and all obtained explanations will have size n. If these concepts are dependent, we get shorter explanations.

Test cases were generated so that $A_1, ..., A_n$ were always independent. They were grouped into 5 groups, S_1–S_5, according to the size of the largest explanation, so that S_i contained all test cases where the largest explanation had length i. Each group contained 10 test cases. So we had 50 test cases in

total.

The test cases were also grouped into another 5 groups, C_1–C_5, according to the second parameter whose influence we wanted to observe: the number of explanations.

The statistics of the created groups can be seen in Table 6.2 (S groups) and Table 6.3 (C groups). Table column names have the following meaning: #: number of test cases; C_m, C_a, C_M: min, average, and max number of explanations; S_m, S_a, S_M: min, average, and max size of the largest explanation;

Set	#	C_m	C_a	C_M	S_m	S_a	S_M
S1	10	1	7	20	1	1	1
S2	10	8	69.5	159	2	2	2
S3	10	47	212.4	479	3	3	3
S4	10	251	417.8	839	4	4	4
S5	10	503	2627	6719	5	5	5

Table 6.2: Statistics of S_1–S_5 groups (test cases divided according to the size of the largest explanation)

Set	#	C_m	C_a	C_M	S_m	S_a	S_M
C1	9	1	4.8	9	1	1.11	2
C2	11	14	51	99	1	2	3
C3	13	111	212.92	299	2	3.15	4
C4	8	359	524.75	839	3	4	5
C5	9	1175	2863	6719	5	5	5

Table 6.3: Statistics of C_1–C_5 groups (test cases divided according to the number of explanations)

Evaluation process

Above, we described what observations and used ontology look like. Next, we specify other properties of the inputs. Timeout was set to 4 hours (14 400 seconds). To replicate the setting of the previous evaluation, we did not allow role assertions in explanations.

We want to find out what effect has allowing negated assertions in explanations. Note that, explanations for generated test cases contain only atomic concept assertions. That means the explanations for a given test case should be the same with or without including negated concept assertion, only search space differs. We also want to compare the efficiency of both the MHS-MXP and the MHS algorithm. Therefore, we created 4 inputs for each test case (observation):

Input 1: using the MHS-MXP algorithm; atomic and negated atomic concept assertions are allowed in explanations.

Input 2: using the MHS algorithm; atomic and negated atomic concept assertions are allowed in explanations.

Input 3: using the MHS-MXP algorithm; only atomic concept assertions are allowed in explanations.

Input 4: using the MHS algorithm; only atomic concept assertions are allowed in explanations.

From this, we get 200 inputs in total. Each input was run 5 times and the average runtime was calculated.

6.1.2 Results

We illustrated the results of Experiment 1 in Figure 6.1 and Figure 6.2. Figure 6.1 contains the results of the inputs with both atomic and negated atomic concept assertions allowed in explanations and Figure 6.2 contains the results of the inputs with only atomic concept assertions in explanations. Both figures have the same structure. Results obtained by using the MHS algorithm are shown on the left, and MHS-MXP results are on the right. Next, we can see the results of groups S_1–S_5 shown at the top and the results for groups C_1–C_5 below. In order to measure how efficiently the algorithms obtain explanations, we displayed the average time (y-axis) in which all explanations

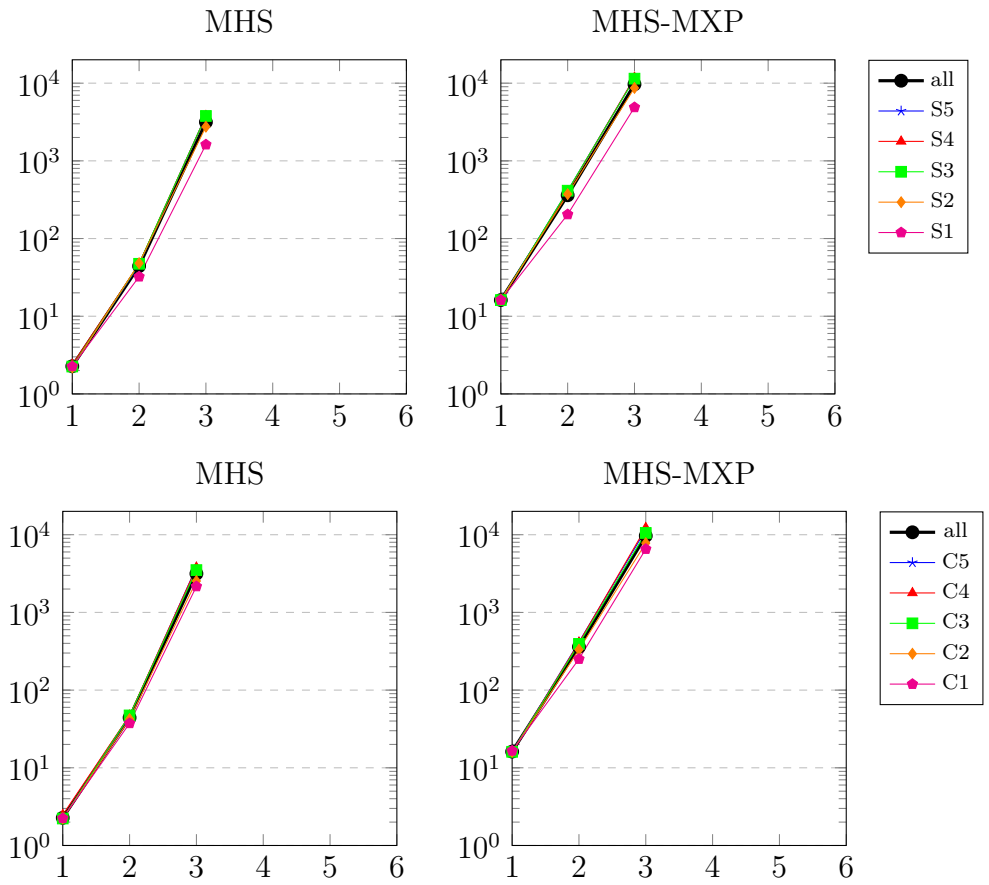


Figure 6.1: **Unfavourable inputs:** Results of inputs with allowed negation in explanations

of a given length (x-axis) are guaranteed to be found, individually for each group. In the case of the MHS algorithm, this corresponds to the fact that it fully explored the HS-tree to depth x, and in the case of MHS-MXP, a full exploration of the HS-tree to depth $x - 1$. If a group of inputs completes the search, a line with the time of completion is drawn further to the right to show that essentially the entire space of possible explanations has been explored. If, on the other hand, a timeout occurred for more than 1/3 of the inputs from a group during the search in a certain depth, the record will be omitted from the graph and the line segment of the group will end at the previous fully searched depth.

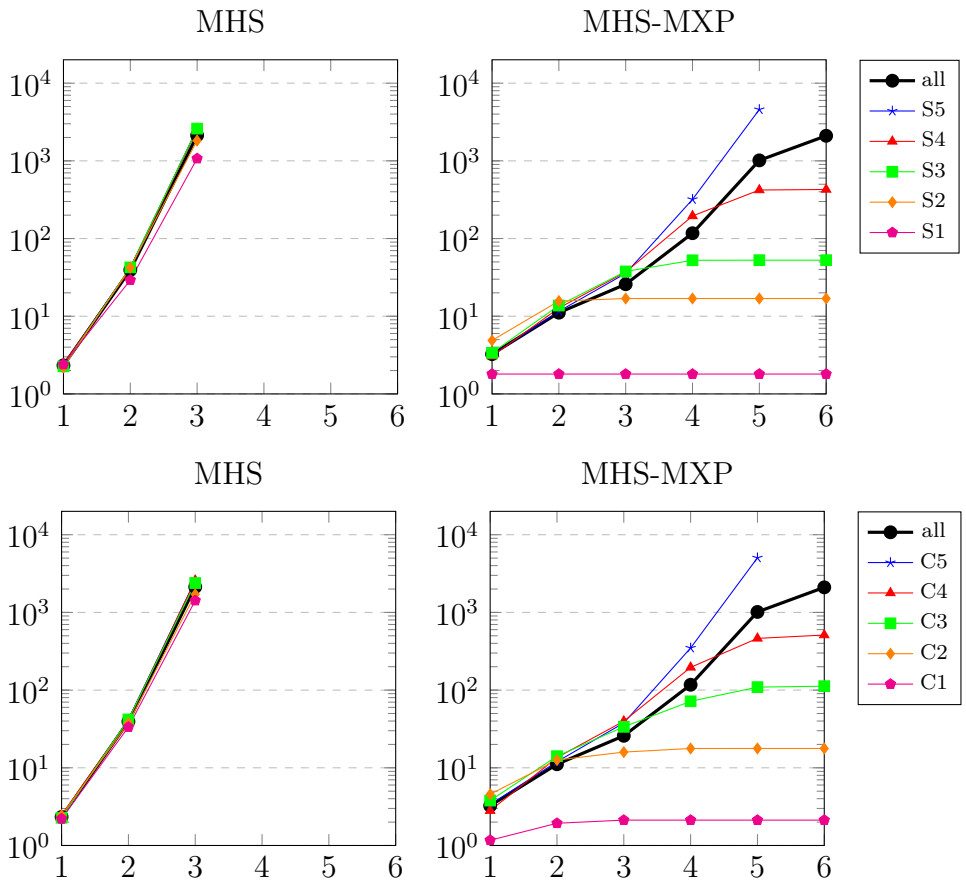


Figure 6.2: **Favourable inputs:** Results of inputs without allowed negation in explanations

Unfavourable inputs: In the graphs from Figure 6.1, we can observe that time grows exponentially with the increasing depth and size of potential explanations that are explored. Both algorithms managed to find all explanations up to size 3. Neither group of inputs managed to complete the search before reaching the timeout. There are small differences between input groups S and C. It is interesting, however, that even within S groups there are only small differences between S_1 – S_5 (the same applies to C groups).

In this case, the MHS algorithm was faster and outperformed the MHS-MXP algorithm. This case, therefore, turns out to be very **un-**

favourable for MHS-MXP. This is caused by conflicts within abducibles that are created by allowing negation in explanations. Abducibles conflicts fill the search space with inconsistent explanations of the type $\{a\colon A, a\colon \neg A\}$, where a is an individual and A is an atomic concept. This causes the MHS-MXP algorithm to constantly find undesired explanations and thus does not know, that it has already found all the desired ones and can end the search, e.g. for S_1 inputs.

The only advantage of the MHS-MXP algorithm in such a case is that it may have managed to find some explanations that have a size greater than 3.

Favourable inputs: When testing inputs without negation allowed in explanations, we can fully observe the advantages of MHS-MXP compared to MHS.

The behaviour of MHS is not very different from the previous case. All inputs reached the timeout, the computation was terminated during the 4th level of the HS-tree and only explanations up to size 3 were obtained.

With MHS-MXP, we achieved significantly better results. Therefore, we can call inputs of this type **favourable** for the MHS-MXP algorithm. Almost all inputs end before reaching timeout. The only exceptions are groups C_5 and S_5, where approximately half of the inputs ended with a timeout.

We observe that the computation time has a certain correlation with the size of the largest explanation of the given input. The S_1 group ends almost immediately and each following group S_2–S_5 requires more computation time than the previous one. This confirms that MHS-MXP is more efficient on inputs with a shorter size of the largest explanation.

The results of MHS-MXP are very similar also for groups C_1–C_5. This indicates that MHS-MXP has a greater advantage on inputs with a smaller number of explanations. However, this could be possibly caused

by the way the inputs were generated, since the inputs with a small size of the largest explanation tended to have a small number of explanations. Likewise, the ones with the largest size were the ones with the largest number of explanations. Therefore, the influence of this parameter can be further explored in the future in other experiments.

6.2 Experiment 2

In the second experiment, we wanted to test the MHS-MXP algorithm on real-world ontologies, which would have different sizes (axiom counts) and structure complexity. We were deeply inspired by the work of Koopmann et al. [14], which gave us suggestions on which ontologies to use and how we can generate observations and abducibles.

In this part, we will not allow explanations with negated assertions since, in Experiment 1, we showed their negative influence on the MHS-MXP algorithm. We will focus on favourable inputs, which allow only positive assertions in explanations.

6.2.1 Methodology

Ontologies

The ontologies that we used in the experiment were chosen from *ORE 2015 Reasoner Competition Corpus*[1]. This corpus was created to compare the capabilities of reasoners with various reasoning tasks.

Reasoner corpus contains 1920 ontology files. For the needs of this experiment, the sufficient number of ontologies is 20. Therefore, we decided to explore these ontologies, find out their basic metrics and filter out the ones that are not suitable for us.

We found that among these ontologies there are some that are inconsistent and ontologies for which the consistency check takes more than a minute.

[1] `https://zenodo.org/record/18578#.Y3tygXbMJPb`

These kinds of ontologies are clearly unsuitable for us. For inconsistent ontologies, there is no point in looking for explanations and the second case is not suitable because, in both algorithms, the consistency check is called a considerable number of times.

We also applied other criteria in order to get only those ontologies that really interest us. First, we filtered out excessively large ontologies that have more than 10 000 logical axioms. We also limited the number of individuals to a maximum of 100. The last limitation was the number of assertions to be at least 100 (they will be used for generating the observations). Lastly, we modified the ontologies and removed data properties from them to solve the problem with the `OWLKnowledgeExplorerReasoner` interface mentioned in Section 5.1.1. From the resulting group we randomly chose 20 of them from uniform distribution according to their logical axiom count.

Moreover, we want such test cases which are non-trivial. So we required that the input ontology did not entail the observation from the input, otherwise, there would be *nothing to explain*. We also required that there is a conflict between abducibles and input ontology united with negated observation. Without such a conflict, we would immediately know that there are *no explanations*. If it was not possible to generate input over the given ontology which meets these conditions, we replaced the ontology with another one.

The basic metrics of chosen ontologies are shown in Table 6.4.

Observations generating

For each of the selected 20 ontologies, we generated 3 observations:

- a single observation (1 assertion)
- a multiple observation of size 5 (5 assertions)
- a multiple observation of size 10 (10 assertions)

Role assertions were also allowed in observations.

Unlike in Experiment 1, we wanted to try another method of generating observations which would be less random. Therefore, we used the method

ontology id	concepts	roles	individuals	logical axioms
11296	28	39	42	286
16814	28	39	42	287
5204	71	43	88	364
6859	170	14	81	434
3884	183	40	91	460
1430	46	19	93	483
14883	125	56	23	562
8460	128	58	23	567
8578	224	67	66	569
15291	224	67	66	569
12566	170	106	75	605
1784	194	59	23	607
8362	26	38	100	627
8975	224	67	82	651
2860	113	76	72	702
155	379	128	82	738
1931	124	75	78	828
4469	131	140	89	960
10807	820	46	66	1232
4796	200	295	39	1582

Table 6.4: The basic metrics of chosen ontologies from *ORE 2015 Reasoner Competition Corpus*.

from the work of Koopmann et al. [14], in which the observation is selected directly from an ontology. In the case of multiple observations, we also wanted to achieve that the selected assertions were related in some way, so we used the **modules extraction** [23] from an ontology.

Module extraction enables the division of an ontology into some parts, called modules. One module then contains axioms that are related to each other.

When generating the observation for an ontology, we proceeded as follows. First, we extracted the modules of the given ontology. Next, we sorted them by size (number of assertions). If the largest module had less than 100 assertions, we united the modules until we got at least 100 assertions. Then, we randomly selected 1, 5 and 10 assertions from these obtained 100+ module

assertions. These selected assertions are our observations.

Finally, it was necessary to modify the ontology for each observation so that it does not contain observation assertions. If the ontology contains observation, the observation would automatically follow from the ontology and all test cases would have nothing to explain. So for each observation, we created its own version of the ontology without assertions from observation.

Abducibles generating

Since the ontologies from Experiment 2 are larger than the LUBM ontology which we have been working with so far, it was important to somehow limit the space in which we are looking for explanations. Therefore we limited the abducibles and chose only 60% of the symbols from the ontology. We also considered only symbols of concepts and individuals in abducibles, since role assertions in explanations are not allowed.

We chose those 60% symbols by selecting them from a uniform distribution according to the number of occurrences in the ontology. In this way, the resulting set of abducibles contains symbols that occur frequently in the ontology, but also symbols that occur there occasionally. This method is also inspired by the work of the work of Koopmann et al. [14].

Evaluation process

We generated a total of 60 observations (3 observation per each of the 20 ontologies). Each observation also has its own set of abducibles defined. Above, we described how do observations, ontologies and abducibles look like. Next, we specify other properties of the inputs.

Timeout was set to 4 hours (14 400 seconds). We did not allow role assertions and negations in explanations. We also limited the maximal depth of the HS-tree to 5. Since multiple observations appear in the inputs, we had to choose the type of relevance that we will use during the experiment. We chose partial relevance (defined in Section 5.2).

We created 2 versions for each observation. In version 1 we used the MHS-MXP algorithm and in version 2 we used the MHS algorithm. From this, we get 120 inputs in total. In the end, we only tested the MHS-MXP inputs. Each MHS-MXP input was run 5 times. We observed how the runs ended (timeout, error or completed search) and how many explanations were found.

6.2.2 Results

Experiment 2 revealed shortcomings of our solver. The proposed inputs were too demanding. The MHS-MXP algorithm could not even reach level 2 of the HS-tree in 4 hours. Because of those circumstances, we did not test inputs with the MHS algorithm, so the comparison of algorithms will not be provided. Nevertheless, we will describe the results of the MHS-MXP algorithm and the observations we reached.

In Table 6.5 we can see how many inputs ended their run in a given way. More than half of the inputs ended with `OutOfMemoryError`. Then a large part ended with a timeout and only a few inputs managed to finish before reaching timeout.

We found that there can be a certain connection between the inputs that end in an error and the number of individuals in a given input ontology. Inputs with ontologies that have up to 50 individuals ended normally. On the other hand, 80% of inputs with ontologies that have over 50 individuals ended with an error.

# error occurred	**# reached timeout**	**# completed search**
34	24	2

Table 6.5: Statistics of how inputs ended.

We also observed that the desired explanations were found for 3 inputs. Note that the inputs were not generated in a way that ensures the desired explanations exist, so other inputs may not even have them. Information about the found explanations is shown in Table 6.6.

ontology id	observation size	explanation count	explanations size
8460	5	7	1
14883	1	1	1
14883	10	4	4

Table 6.6: Statistics of found desired explanations.

Experiment 2 gives us important insights into the future, which concern the design of evaluation inputs and at the same time the technical areas of the solver, which could be improved.

When choosing the input ontologies, we applied many conditions that led to too large ontologies. It seems like we underestimated the number of individuals and did not limit abducibles enough. During the next evaluation, it will definitely be necessary to pay more attention to the selection of ontologies and also abducibles.

We can also look at cases where the inputs ended with an error and examine where the problem occurs.

Conclusion

In this work, our main goal was to improve the existing implementation [4] of the abduction solver, which allows the use of the MHS-MXP algorithm [10] or the well-known MHS algorithm [18] to search for explanations. We dealt with the problem of extracting models, which is essential for the correct functioning of both algorithms and was not performed correctly in the previous version of the solver. Since it is not one of the standard tasks of a DL reasoner, it was unknown whether this problem can be solved while maintaining the black box approach to a DL reasoner. After examining several possible solutions, we have ensured the correct model extraction by using the `OWLKnowledgeExplorerReasoner` interface[1], although it was also necessary to intervene in its JFact reasoner implementation[2] in order to make it work. Subsequently, we enabled role assertions in the explanations. We have presented two ways to define relevance for multiple observation: strict and partial relevance. We have adjusted the location of the log files and added new types that record information about the given run and errors. Subsequently, we fixed a few bugs in the implementation, which caused the algorithm to work incorrectly in some cases.

We performed an empirical evaluation on the new version of the solver, with the goal to compare the MHS-MXP algorithm and the MHS algorithm. The evaluation consisted of two experiments. Experiment 1 consisted in repeating the experiment from previous work [4], this time with the correct

[1] `https://owlcs.github.io/owlapi/apidocs_4/org/semanticweb/owlapi/reasoner/knowledgeexploration/OWLKnowledgeExplorerReasoner.html`

[2] `https://github.com/boborova3/jfact/tree/test4`

extraction of models and inclusion of the unfavourable inputs for MHS-MXP. On unfavourable inputs, MHS-MXP achieved worse results than MHS. However, on favourable inputs, which excluded negated assertions from explanations, MHS-MXP was significantly better. In Experiment 2, we tested MHS-MXP on different real-world ontologies for the first time. The experiment helped reveal the shortcomings of our solver and the MHS-MXP algorithm for large input ontologies with many individuals. From the selected sample of ontologies, most inputs ended with a timeout or `OutOfMemoryError`. Its results can be valuable for future development.

In the future, more attention can be paid to optimization, for example, model reusing is probably not utilised to its full potential. It also would be useful to fix the JFact reasoner implementation of the `OWLKnowledgeExplorerReasoner` interface, which still contains some bugs. There are also many options for evaluation. For example, repeating Experiment 2 with greater variability of inputs by including less difficult ones as well. When generating inputs, we could try to limit abducibles (mainly individuals) more strictly. Another option could be examining Experiment 1 and finding the boundary where MHS-MXP still achieves better results.

Bibliography

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

[2] F. Baader, I. Horrocks, C. Lutz, and U. Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.

[3] F. Baader, I. Horrocks, and U. Sattler. Description logics. In F. van Harmelen, V. Lifschitz, and B. W. Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 135–179. Elsevier, 2008.

[4] I. Balintová. Heuristic optimization of the MHS-MXP algorithm. Master's thesis, Comenius University in Bratislava, 2022.

[5] S. Bechhofer, R. Möller, and P. Crowther. The DIG description logic interface. In D. Calvanese, G. D. Giacomo, and E. Franconi, editors, *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy September 5-7, 2003*, volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[6] I. Dickinson. Implementation experience with the DIG 1.1 specification. *Hewlett Packard, Digital Media Sys. Labs, Bristol, Tech. Rep. HPL-2004-85*, 2004.

[7] C. Elsenbroich, O. Kutz, and U. Sattler. A case for abductive reasoning over ontologies. In B. C. Grau, P. Hitzler, C. Shankey, and E. Wallace,

editors, *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[8] N. Guarino, D. Oberle, and S. Staab. What is an ontology? In S. Staab and R. Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 1–17. Springer, 2009.

[9] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.*, 3(2-3):158–182, 2005.

[10] M. Homola, J. Pukancová, I. Balintová, and J. Boborová. Hybrid mhs-mxp abox abduction solver: First empirical results. In O. Arieli, M. Homola, J. C. Jung, and M. Mugnier, editors, *Proceedings of the 35th International Workshop on Description Logics (DL 2022) co-located with Federated Logic Conference (FLoC 2022), Haifa, Israel, August 7th to 10th, 2022*, volume 3263 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[11] M. Homola, J. Pukancová, J. Gablíková, and K. Fabianová. Merge, explain, iterate. In S. Borgwardt and T. Meyer, editors, *Proceedings of the 33rd International Workshop on Description Logics (DL 2020) co-located with the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020), Online Event [Rhodes, Greece], September 12th to 14th, 2020*, volume 2663 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

[12] M. Horridge and S. Bechhofer. The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

[13] M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang. The manchester OWL syntax. In B. C. Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA,*

November 10-11, 2006, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[14] P. Koopmann, W. Del-Pinto, S. Tourret, and R. A. Schmidt. Signature-based abduction for expressive description logics. In D. Calvanese, E. Erdem, and M. Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 592–602, 2020.

[15] A. Krisnadhi and P. Hitzler. A tableau algorithm for description logics with nominal schema. In M. Krötzsch and U. Straccia, editors, *Web Reasoning and Rule Systems - 6th International Conference, RR 2012, Vienna, Austria, September 10-12, 2012. Proceedings*, volume 7497 of *Lecture Notes in Computer Science*, pages 234–237. Springer, 2012.

[16] C. S. Peirce. Deduction, Induction, and Hypothesis. *Popular Science Monthly*, 13:470–482, 1878. From the Commens Bibliography | `http://www.commens.org/bibliography/journal_article/peirce-charles-s-1878-deduction-induction-and-hypothesis`.

[17] J. Pukancová and M. Homola. Tableau-based ABox abduction for the $\mathcal{ALCHO}$ description logic. In A. Artale, B. Glimm, and R. Kontchakov, editors, *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017*, volume 1879 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[18] J. Pukancová and M. Homola. Abox abduction for description logics: The case of multiple observations. In M. Ortiz and T. Schneider, editors, *Proceedings of the 31st International Workshop on Description Logics co-located with 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), Tempe, Arizona, US, October 27th - to - 29th, 2018*, volume 2211 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[19] J. Pukancová. *Direct Approach to ABox Abduction in Description Logics*. PhD thesis, Comenius University in Bratislava, 2018.

[20] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987.

[21] S. Rudolph. Foundations of description logics. In A. Polleres, C. d'Amato, M. Arenas, S. Handschuh, P. Kroner, S. Ossowski, and P. F. Patel-Schneider, editors, *Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*, volume 6848 of *Lecture Notes in Computer Science*, pages 76–136. Springer, 2011.

[22] K. M. Shchekotykhin, D. Jannach, and T. Schmitz. Mergexplain: Fast computation of multiple conflicts for diagnosis. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3221–3228. AAAI Press, 2015.

[23] C. D. Vescovo, D. Gessler, P. Klinov, B. Parsia, U. Sattler, T. Schneider, and A. Winget. Decomposition and modular structure of bioportal ontologies. In L. Aroyo, C. Welty, H. Alani, J. Taylor, A. Bernstein, L. Kagal, N. F. Noy, and E. Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2011.

List of Figures

1.1 Clash-free CTree T for $\mathcal{K}_4$ 25

2.1 Pruned HS-tree for $F = \{\{1,2,3\}\{3,4\},\{5,6\}\}$ 35
2.2 HS-tree for ABox abduction problem in Example 2.1.1 48

6.1 **Unfavourable inputs:** Results of inputs with allowed negation in explanations 80
6.2 **Favourable inputs:** Results of inputs without allowed negation in explanations 81