FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
OF THE COMENIUS UNIVERSITY IN BRATISLAVA

OPTIMIZATION OF AN ABDUCTIVE REASONER FOR DESCRIPTION LOGICS

Master thesis

2019 Katarína Fabianová

FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

OF THE COMENIUS UNIVERSITY IN BRATISLAVA

OPTIMIZATION OF AN ABDUCTIVE REASONER FOR DESCRIPTION LOGICS

Master thesis

Study program:	Applied informatics
Field of study:	2511 Applied informatics
School department:	Department of Applied Informatics
Adviser:	Mgr. Júlia Pukancová, PhD.
Consultant:	RNDr. Martin Homola, PhD.

Bratislava 2019 **Katarína Fabianová**

Acknowledgements

Abstract

Key words:

Abstrakt

Kľúčové slová:

Contents

List of Figures

Introduction

Description logics

Description logics (DLs) are a family of knowledge representation formalism. Each description logic has different expressivity. Every expressivity is expressed with a unique set of constructors. We are going to work with $\mathcal{ALC}$, $\mathcal{EL}$ and $\mathcal{EL}$++ DL. Each description logic has its own syntax and semantics. In this chapter we will introduce syntax and semantics (Rudolph, 2011) for each DL that we will be working with.

$\mathcal{ALC}$ DL

$\mathcal{ALC}$ DL is a DL which is more expressive than $\mathcal{EL}$ and $\mathcal{EL}$++ DL. $\mathcal{ALC}$ is less expressive than many other DLs. $\mathcal{ALC}$ stands for Attributive (Concept) Language with Complements. It means that not only complement of atomic concept is allowed but also a complement of complex concept is allowed.

$\mathcal{ALC}$ Syntax

$\mathcal{ALC}$ description logic consists of three mutually disjoint sets. These sets represent whole vocabulary that is used by $\mathcal{ALC}$ DL.

Definition 1.1.1 ***DL vocabulary***

$$\textbf{\textit{Set of individuals:}}\ N_I = \{a, b, c...\}$$

$$\textbf{\textit{Set of concepts:}}\ N_C = \{A, B, C...\}$$

$$\textbf{\textit{Set of roles:}}\ N_R = \{R_1, R_2, R_3, ...\}$$

Example 1.1.1 ***DL vocabulary***

$$N_I = \{jack, john, jane\}$$

$$N_C = \{Person, Mother, Father\}$$

$$N_R = \{hasChild, likes, owns\}$$

$\mathcal{ALC}$ DL deals with individuals and concepts. An individual is a concrete instance of a concept. Concept is a class that defines some entity. Concept can be atomic or complex. Atomic concept is not constructed with any constructor. On the contrary complex concept is created from constructors and other concepts.

Definition 1.1.2 ***Complex concept***

Concepts are recursively constructed as the smallest set of expressions of the forms:

$$C, D ::= A|\neg C|C \sqcap D|C \sqcup D|\exists R.C|\forall R.C$$

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

Example 1.1.2 ***Complex concept***

$$\neg Mother$$

$$Mother \sqcup Father$$

$$\exists hasChild.Person$$

$$\forall likes.Food$$

Complex concept uses following constructors: $\neg$, $\sqcup$, $\sqcap$, $\exists$ and $\forall$. Constructor $\neg$ is negation, constructor $\sqcup$ is or and constructor $\sqcap$ is and. Constructors $\exists$ is existential restriction and $\forall$ is called value restriction.

There are two concepts that are always in ontology. $\top$ (top) stays for everything. Each concept belongs under $\top$ which means that each concept is on left side of subsumption if on right side is only $\top$. Second concept is $\bot$ (bottom) and it stays for nothing which means that each concept is on the right side of subsumption if there is only $\bot$ on the left side. Formally these two concepts can be written as follows:

$$\top \equiv A \sqcup \neg A$$

$$\bot \equiv A \sqcap \neg A$$

$\mathcal{ALC}$ description logic uses axioms in order to model some situation. Ontology (Fitz-Gerald and Wiggins, 2010) is used to formally describe these axioms. The purpose of ontology is to describe relationships between entities in a formal language. Every ontology has its own knowledge base. Knowledge base is a set of TBox axioms and ABox axioms. Ontology is described by knowledge base.

Definition 1.1.3 ***Knowledge base***

Knowledge base ($\mathcal{KB}$) is an ordered pair of TBox $\mathcal{T}$ and ABox $\mathcal{A}$.

Example 1.1.3 ***Knowledge base***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \\ jack : Academician \\ jane : Scientist \\ john : Professor \end{array} \right\}$$

There are two types of axioms, the first is TBox and the second one is ABox. In TBox the subsumption symbol ($\sqsubseteq$) is used. Let's explain this symbol on an example

$Mother \sqsubseteq Parent$. $Mother$ is always a $Parent$ but $Parent$ does not always have to be a $Mother$. TBox represents axioms that model ontology. Each axiom explains the relationship between entities in this axiom.

Definition 1.1.4 ***TBox***

A TBox $\mathcal{T}$ is a finite set of GCI axioms ϕ of the form:

$$\phi ::= C \sqsubseteq D$$

where C, D are any concepts.

Example 1.1.4 ***TBox***

$$\mathcal{T} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \end{array} \right\}$$

ABox on the other side does not model ontology but creates a database of facts. It contains set of assertion axioms that can be called facts. Fact is a direct assertion of an individual to a concept.

Definition 1.1.5 ***ABox***

An ABox $\mathcal{A}$ is a finite set of assertion axioms ϕ of the form:

$$\phi ::= a : C | a, b : R$$

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

Example 1.1.5 ***ABox***

$$\mathcal{A} = \left\{ \begin{array}{c} jack : Academician \\ jane : Scientist \\ john : Professor \end{array} \right\}$$

To have a better understanding of $\mathcal{ALC}$ DL we can translate sentences into $\mathcal{ALC}$ description logic.

Everybody who is sick, is not happy.

$$Sick \sqsubseteq \neg Happy$$

Cat and dogs are animals.

$$Cat \sqcup Dog \sqsubseteq Animal$$

Every person owns a house.

$$Person \sqsubseteq \exists owns.House$$

$\mathcal{ALC}$ Semantics

An interpretation is a pair of a domain and an interpretation function. The domain is a set of values that represents concepts in the interpretation. The result of interpretation function is different for individuals, concepts and roles. If we use the interpretation function on an individual the result is an element from the domain. If we use it on the concepts the result is a set of elements from the domain. If we use it on the roles the result is a set of pairs of elements from the domain.

Definition 1.1.6 ***Interpretation***

An interpretation of a given knowledge base $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ *is a pair* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ *which contains a domain* $\Delta^{\mathcal{I}}$ *and an interpretation function* $\cdot^{\mathcal{I}}$*. Domain can not be empty.*

The interpretation function is following:

$$a^{\mathcal{I}} \in \Delta^{\mathcal{I}} \ \forall a \in N_I$$

$$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \ \forall A \in N_C$$

$$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \ \forall R \in N_R$$

The interpretation of complex concepts is recursively defined:

$$\neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$$

$$C \sqcap D^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$$

$$C \sqcup D^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$$

$$\exists R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \exists y \in \Delta^{\mathcal{I}} : \langle x, y \rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$$

$$\forall R.C^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} | \forall y \in \Delta^{\mathcal{I}} : \langle x, y \rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$$

Example 1.1.6 ***Interpretation***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \\ jack : Academician \end{array} \right\}$$

$$\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$$

$$\Delta^{\mathcal{I}} = \{P, S, A, AP\}$$

$$Professor^{\mathcal{I}} = \{P\}$$

$$Scientist^{\mathcal{I}} = \{S\}$$

$$Academician^{\mathcal{I}} = \{A\}$$

$$AssocProfessor^{\mathcal{I}} = \{AP\}$$

$$jack^{\mathcal{I}} = A$$

The interpretation satisfies an axiom according to its type. We know three types of axioms. The first is an axiom from TBox, the second one is an assertion axiom to a concept and the third one is an assertion to a role.

Definition 1.1.7 ***Satisfaction*** $\models$

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

$$\mathcal{I} \models C \sqsubseteq D \textit{ iff } C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$$

$$\mathcal{I} \models a : C \textit{ iff } a^{\mathcal{I}} \in C^{\mathcal{I}}$$

$$\mathcal{I} \models a, b : R \textit{ iff } \langle a^{\mathcal{I}}, \in b^{\mathcal{I}} \rangle \in R^{\mathcal{I}}$$

Finding a model (an interpretation satisfying $\mathcal{KB}$) is crucial for consistency checking. If we find at least one model the knowledge base $\mathcal{KB}$ is consistent. We can have more models for one knowledge base. In Example 1.1.7 there are two showed models but there can be more models.

Definition 1.1.8 ***Model***

An interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ is a model of a DL knowledge base $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ iff $\mathcal{I}$ satisfies every axiom in TBox $\mathcal{T}$ and ABox $\mathcal{A}$.

Example 1.1.7 ***Model***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ \\ AssocProfessor \sqsubseteq Professor \\ \\ jack : Academician \end{array} \right\}$$

$$\mathcal{I}_1 = \{\Delta^{\mathcal{I}} = \{P, S, A, AP\},$$

$$Professor^{\mathcal{I}} = \{P\}, Scientist^{\mathcal{I}} = \{S\}, Academician^{\mathcal{I}} = \{A\}, AssocProfessor^{\mathcal{I}} = \{AP\}$$

$$jack^{\mathcal{I}} = A\}$$

$$\mathcal{I}_2 = \{\Delta^{\mathcal{I}} = \{A\},$$

$$Professor^{\mathcal{I}} = \{A\}, Scientist^{\mathcal{I}} = \{A\}, Academician^{\mathcal{I}} = \{A\}, AssocProfessor^{\mathcal{I}} = \{A\}$$

$$jack^{\mathcal{I}} = A\}$$

Definition 1.1.9 ***Consistency***

A knowledge base $\mathcal{KB}$ is consistent iff $\mathcal{KB}$ has at least one model $\mathcal{I}$.

We are familiar with more decision problems, satisfiability, subsumption ($\sqsubseteq$), equivalence ($\equiv$) and disjointness. Satisfiability means that concept is satisfiable in regard to $\mathcal{KB}$ if we can find such a model of a knowledge base for which holds that interpretation of that concept is not empty. Subsumption between two concepts must hold that interpretation of left-sided concept is a proper subset of interpretation of right-sided concept in each possible model of a knowledge base. Equivalence is similar to subsumption but the difference is that the interpretations of both concepts must be equal in each possible model of a knowledge base. Disjointness means that intersection of the interpretations of both concepts must be an empty set in each possible model of a knowledge base.

Definition 1.1.10 ***Decision problems***

Given a DL $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$, and two concepts C,D, we say that:

- *C is satisfiable w.r.t. $\mathcal{KB}$ iff there is such a model $\mathcal{I}$ of $\mathcal{KB}$ for which holds that $C^{\mathcal{I}} \neq \emptyset$;*
- *C is subsumed by D w.r.t. $\mathcal{KB}$ (denoted $\mathcal{KB} \models C \sqsubseteq D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{KB}$;*
- *C and D are equivalent w.r.t. $\mathcal{KB}$ (denoted $\mathcal{KB} \models C \equiv D$) iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ in every model $\mathcal{I}$ of $\mathcal{KB}$;*
- *C and D are disjoint w.r.t. $\mathcal{KB}$ iff $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ in every model $\mathcal{I}$ of $\mathcal{KB}$.*

If $\mathcal{KB} = \emptyset$ then we say that decision problems (satisfiability, subsumption, equivalence and disjointness) of concepts are defined in general by definition and we use notation such as we omit $\mathcal{KB} \models$ from the notation.

DL Tableau algorithm

Following definitions (Baader et al., 2003) describe rules and terms that are used by this algorithm. DL Tableau algorithm proves satisfiability of a concept or checks consistency of a given knowledge base. By proving satisfiability the input for the algorithm is concept C and TBox $\mathcal{T}$. By checking consistency the input for the algorithm is a knowledge base $\mathcal{KB}$. In both versions of this algorithm holds that each concept must be in NNF (negation normal form). Negation normal form is such a form that each negation is pushed in front of atomic concept inside a complex concept.

Definition 1.2.1 ***NNF***

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

Example 1.2.1 ***NNF***

$$nnf(\neg(C \sqcap D)) = \neg C \sqcup \neg D$$

$$nnf(\neg(C \sqcup D)) = \neg C \sqcap \neg D$$

$$nnf(\neg \exists R.C) = \forall R.\neg C$$

$$nnf(\neg \forall R.C) = \exists R.\neg C$$

DL Tableau algorithm creates a CTree (Completion tree). CTree proves whether model exists or not.

Definition 1.2.2 ***Completion tree***

A completion tree (CTree) is a triple T = (V, E, $\mathcal{L}$) where (V, E) is a tree and $\mathcal{L}$ is a labeling function which means that: $\mathcal{L}(x)$ is a set of concepts $\forall x \in V$ and $\mathcal{L}(\langle x, y\rangle)$ is a set of roles $\forall \langle x, y\rangle \in E$.

At first the algorithm creates node and initializes it with the input concept. Then the algorithm applies tableau rules for given DL. If the algorithm applies each rule until any rule can be applied and it finds no clash then result is that concept C is satisfiable w.r.t. $\mathcal{T}$.

Definition 1.2.3 ***Successor, R-successor***

Given a CTree T = (V,E, $\mathcal{L}$) and $x, y \in V$ we say that:

- *y is a successor of x iff $\langle x, y\rangle \in E$*
- *y is an R-successor of x iff $\langle x, y\rangle \in E$ and $R \in \mathcal{L}(\langle x, y\rangle)$.*

Definition 1.2.4 ***Clash***

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

Example 1.2.2 ***Clash***

$$\mathcal{L}(s_0) = \{C, D, \neg D\}$$

Definition 1.2.5 ***Clash-free CTree***

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

There can be a situation that some concept can lead to infinite looping and algorithm would never stop. An example for that is: $Person \sqsubseteq \exists hasParent.Person$. That is why the algorithm uses Blocking rule.

Definition 1.2.6 ***Blocking***

Given a CTree $T = (V, E, \mathcal{L})$ a node $x \in V$ is blocked if it has an ancestor y such that: either $\mathcal{L}(x) \subseteq \mathcal{L}(y)$ or y is blocked.

DL Tableau algorithm proves concept satisfiability in regard of TBox. Input for DL Tableau algorithm is concept C and TBox $\mathcal{T}$. Output is a boolean value which is true if concept C is satisfiable w.r.t. $\mathcal{T}$, false otherwise.

Definition 1.2.7 ***Algorithm - Concept satisfiability***

Input: *concept C and $\mathcal{T}$ in NNF*

Output: *answers whether concept C is satisfiable w.r.t. $\mathcal{T}$ or not*

Steps:

1. *Initialize a new CTree $T := (\{s_0\}, \emptyset, \{s_0 \rightarrow \{C\}\})$;*

2. *Apply tableau rules for TBoxes while at least one rule is applicable;*

3. *Answer "C is satisfiable w.r.t. $\mathcal{T}$" if T is clash-free. Otherwise answer "C is unsatisfiable w.r.t. $\mathcal{T}$".*

Definition 1.2.8 ***$\mathcal{ALC}$ tableau rules for TBoxes***

- $\sqcap - rule$: *if* $C_1 \sqcap C_2 \in \mathcal{L}(x)$ *and* $x \in V$ *and* $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$ *and x is not blocked then* $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$

- $\sqcup - rule$: *if* $C_1 \sqcup C_2 \in \mathcal{L}(x)$ *and* $x \in V$ *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* $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_2\}$

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

- $\exists - rule$: *if* $\exists R.C \in \mathcal{L}(x)$ *and* $x \in V$ *with no R-successor y and* $C \in \mathcal{L}(y)$ *and x is not blocked then* $\mathcal{V} = \mathcal{V} \cup \{z\}, \mathcal{L}(z) = \{C\}$ *and* $\mathcal{L}(\langle x, z\rangle) = \{R\}$

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

Knowledge base is a pair of a TBox and an Abox and yet only a TBox was mentioned in connection with the DL Tableau algorithm. If we also have a given ABox the algorithm must be modified. The algorithm checks no longer concept satisfiability w.r.t. $\mathcal{T}$ but checks consistency of a given $\mathcal{KB}$. The same condition holds for the result as in proving concept satisfiability. If there is a found clash the knowledge base is not consistent otherwise the knowledge base is consistent. Difference between the first version of this algorithm is that here are used named nodes. Name of the node is an instance of a concept that comes from ABox.

Definition 1.2.9 ***Algorithm - Consistency checking***

Input: $\mathcal{KB} = (\mathcal{T}, \mathcal{A})$ *in NNF*

Output: *answers whether* $\mathcal{KB}$ *is consistent or not*

Steps:

1. *Initialize a CTree as follows:*
   - $V := \{a \mid \textit{individual } a \textit{ occurs in } \mathcal{A}\}$;
   - $E := \{\langle a, b\rangle \mid a, b : R \in \mathcal{A} \textit{ for some role } R\}$;
   - $\mathcal{L}(a) := \{nnf(E) \mid a : E \in \mathcal{A}\}$ *for all* $a \in V$;

     $\mathcal{L}(\langle a, b\rangle) := \{R \mid a, b : R \in \mathcal{A}\}$ *for all* $\langle a, b\rangle \in E$;
2. *Apply tableau rules for TBoxes while at least one rule is applicable;*
3. *Answer "*$\mathcal{KB}$ *is consistent" if T is clash-free. Otherwise answer "*$\mathcal{KB}$ *is inconsistent".*

Let's introduce an example where we know a knowledge base $\mathcal{KB}$. DL Tableau algorithm checks if knowledge base is consistent or not.

Example 1.2.3 ***(DL Tableau algorithm)***

$$\mathcal{KB} = \left\{ \begin{array}{c} Student \sqsubseteq \exists attends.MathCourse \\ MathCourse \sqsubseteq Course \\ jack : Student \end{array} \right\}$$

$$\text{jack} \xrightarrow{\text{attends}} m_1$$

$\mathcal{L}(jack) = \{Student, \neg Student \sqcup \exists attends.MathCourse, \exists attends.MathCourse, \neg Logic \sqcup Course, \neg MathCourse\}$

$\mathcal{L}(m_1) = \{MathCourse, \neg Student \sqcup \exists attends.MathCourse, \neg Student, \neg MathCourse \sqcup Course, Course\}$

The algorithm starts with the ABox assertion axioms. It selects the axiom $jack : Student$. At first it creates a node $jack$ with its label $\mathcal{L}(jack)$. From the assertion axiom $jack : Student$ algorithm knows that jack is a Student and adds the concept $Student$ to the label $\mathcal{L}(jack)$. Then the algorithm is choosing another axiom from the knowledge base. There are no more assertion axioms, so it starts with the axioms from TBox. It takes $Student \sqsubseteq \exists attends.MathCourse$. It must convert it into the negation normal form which is $\neg Student \sqcup \exists attends.MathCourse$. The algorithm applies $\sqcup - rule$. According to this rule it must choose $\exists attends.MathCourse$, otherwise there would be a clash. Now it has to apply $\exists - rule$ so a new node l_1 will be created with its label $\mathcal{L}(m_1)$. Concept $MathCourse$ will be added into this label. Let's go back to the $jack$ node. The algorithm has to finish adding all axioms of TBox form. It will add the second axiom from the knowledge base in negation normal form as $\neg MathCourse \sqcup Course$. Now it does not matter which one algorithm chooses if $Course$ or $\neg MathCourse$ because any of these two does not cause a clash. Now the algorithm goes back to the m_1 node and continues to add axioms from the knowledge base similarly as in the node $jack$. It starts with axiom $\neg Student \sqcup \exists attends.MathCourse$, it will choose the first one $\neg Student$. Then continues with axiom $\neg MathCourse \sqcup Course$. It can not choose $\neg MathCourse$ because there would be immediately a clash. So it chooses $Course$.

The created tree contains two nodes ($jack$ and m_1) and one edge between them. The edge represents that $jack$ attends some MathCourse m_1. There is not any instance of a concept $MathCourse$ so the node m_1 does not represent a concrete instance of the concept $MathCourse$ but represents some instance. There is no clash in both nodes so the answer of the DL Tableau algorithm is that the given knowledge base $\mathcal{KB}$ is consistent.

Abduction

Generally in logic we are familiar with three ways of thinking. Deduction, induction and abduction. The most known and natural for humans is probably deduction. All three ways are dealing with the following parts: theory, data and effect. In the description logic we can translate the theory as a knowledge base, data as the explanations and the effect as an observation.

By deduction is known a knowledge base and the explanations, the observation is missing and the goal is to deduce the missing observation. By induction is known an observation and an explanation but the knowledge base is unknown. By abduction is known a knowledge base and an observation but the explanation is a subject of searching. All definitions are from article by Pukancová and Homola (2017).

ABox abduction

In the description logics abduction is used when we are not familiar with the explanation $\mathcal{E}$ but we know a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$. It is important to know that we are looking for minimal explanations. A minimal explanation is such an explanation that it does not exist any other explanation that would be a subset of that minimal explanation.

Definition 2.1.1 ***(Abduction)***

Given a knowledge base $\mathcal{KB}$ and an observation $\mathcal{O}$, an abductive explanation is such an explanation $\mathcal{E}$ that satisfies $\mathcal{KB} \cup \mathcal{E} \models \mathcal{O}$.

Definition 2.1.2 ***(Correct explanation)***

$$\mathcal{E} \textit{ is consistent if } \mathcal{E} \cup \mathcal{KB} \not\models \bot;$$

$$\mathcal{E} \textit{ is relevant if } \mathcal{E} \not\models \mathcal{O};$$

$$\mathcal{E} \textit{ is explanatory if } \mathcal{KB} \not\models \mathcal{O}$$

Definition 2.1.3 ***(Minimal explanation)***

Minimal explanation is such an explanation that it does not exist any other explanation that would be a subset of this explanation.

For better understanding of what ABox abduction is let's introduce a few examples. The first example is easy, the searched explanation is obvious but it will demonstrate the problem. The second example is not so obvious that is why we have to use an algorithm to compute the solution. For computing the solution we use the **Minimal Hitting Set algorithm**.

Example 2.1.1 ***(ABox Abduction - Emotion)***

$$\mathcal{KB} = \left\{ \; Sick \sqsubseteq \neg Happy \; \right\}$$

$$\mathcal{O} = \left\{ \; mary : \neg Happy \; \right\}$$

In this example we are searching the explanations. In this easy assignment it is obvious that what we are looking for is that Mary is sick. If Mary is not happy she must be sick. Formally written solution to this abduction problem is the following:

$$\mathcal{E} = \left\{ \begin{array}{c} mary : Sick \end{array} \right\}$$

Example 2.1.2 ***(ABox Abduction - Academy)***

$$\mathcal{KB} = \left\{ \begin{array}{c} Professor \sqcup Scientist \sqsubseteq Academician \\ AssocProfessor \sqsubseteq Professor \end{array} \right\}$$

$$\mathcal{O} = \left\{ \begin{array}{c} jack : Academician \end{array} \right\}$$

If $jack$ is an Academician he must be a Professor or a Scientist. If he is a Professor he must be also an AssocProfessor. This is an oversimplified explanation how we can retrieve correct explanations. For the better introduction into this algorithm we will explain it step by step. We know the observation so we know that jack is an Academician, that is a fact. In our first axiom in TBox it is written that if somebody is an Academician he is also a Professor or a Scientist. He can be both but at least one of them but that we are not able to determine. That is why both explanations are correct. The second axiom claims that if somebody is a Professor he must be also an AssocProfessor. In this part we are not explaining the whole algorithm yet because it will be explained in our next chapter Minimal Hitting Set algorithm but we need to have at least an idea of how it works. So as we already know the correct explanations are following:

$$\mathcal{E}_1 = \left\{ \begin{array}{c} jack : Professor \end{array} \right\}$$

$$\mathcal{E}_3 = \left\{ \ jack : Scientist \ \right\}$$

$$\mathcal{E}_3 = \left\{ \ jack : AssocProfessor \ \right\}$$

Minimal Hitting Set

In this part we will explain the algorithm: Minimal hitting set algorithm. This algorithm is invented by Raymond Reiter (1987). At first we will declare terms that we will be using. The first term is a hitting set. Let's have collection of sets C. Hitting set is such a set that has non-empty intersection with each set of collection C.

Definition 2.2.1 ***Hitting set***

A hitting set for a collection of sets C is a set $H \subseteq \cup_{S \in C}$ such that $H \cap S$ is not an empty set for each $S \in C$.

HS-tree is such a tree that contains nodes with three possible values. The first value is check mark, the second is cross mark and the third is a set. Each edge has its own label that determines a path from the parent to node. Labeled edges determines path from the root to node n. HS-tree must be the smallest tree for the collection of sets C.

Definition 2.2.2 ***HS-tree***

Let C be a collection of sets. An HS-tree T for C is a smallest edge-labeled and node-labeled tree with the following properties:

- *The root is labeled by $\checkmark$ if C is empty. Otherwise the root is labeled by an arbitrary set of C.*

- *For each node n of T, let $H(n)$ be the set of edge labels on the path in T from the root to a node n. The label for n is any set $\sigma \in C$ such that $\Sigma \cap H(n) = \emptyset$, if such a set Σ exists. Otherwise, the label for n is $\checkmark$. If n is labeled by the set Σ, then for each $\sigma \in \Sigma$, n has a successor n_σ joined to n by an edge of labeled by σ.*

The minimal hitting set algorithm generates a HS-tree. It is a breadth-first search algorithm which means that algorithm creates all nodes in one breadth level and then it goes to the next level if possible. The following definition describes rules of how the algorithm behaves.

Definition 2.2.3 ***Minimal Hitting Set algorithm***

- *Generate the pruned HS-tree breadth-first, generating all nodes at any fixed level in the tree before descending to generate the nodes at the next level.*
- *Reusing node labels: If node n has already been labeled by a set $S \in C$ and if n' is a new node such that $H(n') \cap S = \emptyset$, then label n' by S.*
- *Tree prunning:*
  - *If node n is labeled by $\checkmark$ and node n' is such that $H(n) \subseteq H(n')$, then close the node n'. A label is not computed for n' nor are any successor nodes generated.*
  - *If a node n has been generated and node n' is such that $H(n') = H(n)$, then close node n'.*
  - *If nodes n and n' have been labeled by sets S and S' of C, respectively and if S' is a proper subset of S, then for each $\alpha \in S - S'$ mark as redundant the edge from node n labeled by α. A redundant edge, together with the subtree beneath it, may be removed from the HS-tree while preserving the property that the resulting pruned HS-tree will yield all minimal hitting sets for C.*

Reiter's algorithm is used to compute minimal hitting sets. Input is a set of sets. The goal of Reiter's algorithm is to compute minimal hitting sets from this input. It means that the resulting hitting sets will have a non-empty intersection with each set from the input.

Let's show an example with F as the input set and HS as the result .

Example 2.2.1 ***(Minimal Hitting Set)***

$$F = \{\{a, b\}, \{b, c\}, \{a, c\}, \{b, d\}, \{b\}\}$$

$$HS = \{\{a, b\}, \{b, c\}\}$$

As we can observe in this example the first result set has a non-empty intersection with each set of F and the second result set also has an intersection with each set of F. So we can determine that these resulting sets are definitely minimal hitting sets. They are both minimal because there is no other set that is smaller and at the same time has non-empty intersection with each set of F.

For better and easier understanding of this algorithm we can visualize it and explain (Figure 1).

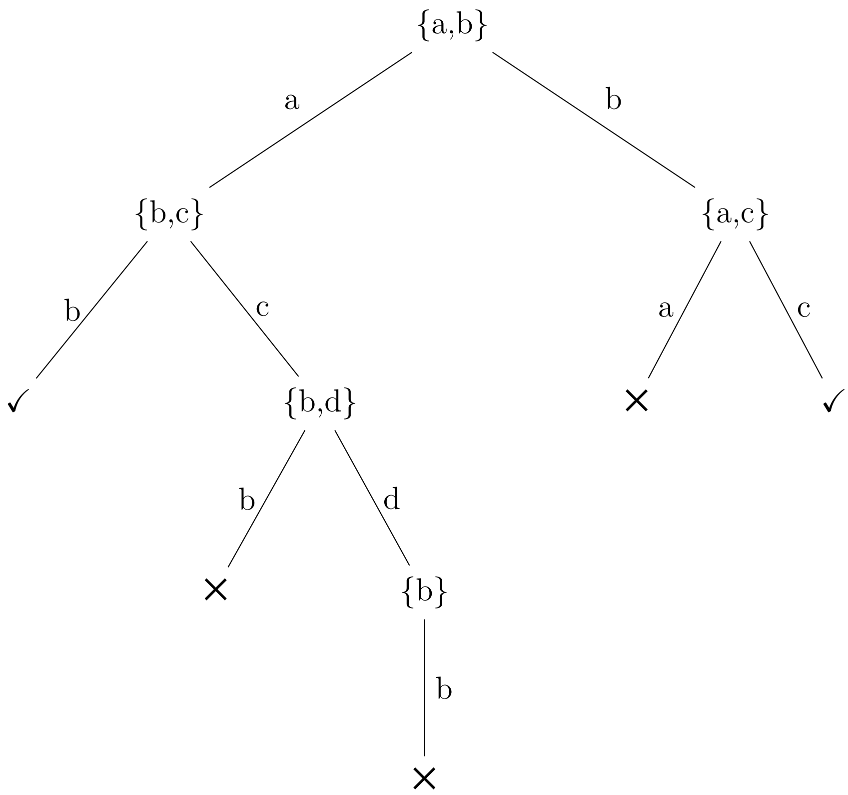

Figure 1: Minimal Hitting Set

An aim is to create a tree where nodes can have three types of a value. A node can be a set from F, check mark or cross mark. Each edge is labeled by one element of some set from F. At first we have to create a root. The root node is the first set $\{a, b\}$ from F. It will have exactly two children because the size of the node label is two.

Now we have to decide what will be the child's node. Let's check the left child. A labeled path from root to the left node is $\{a\}$. If $\{a\}$ has an intersection with each set from F we can add check mark as the node. But that is not our case so we add such a set that has no intersection with the set $\{a\}$, as the node. That set is $\{b, c\}$. Let's continue with the right child of the root. A labeled path from this node to the root is $\{b\}$. That does not have non-empty intersection with each set of F so we add set $\{a, c\}$ as node. We continue breadth-first with $\{b, c\}$ node. Left child has the path labeled by $\{a, b\}$ which has an intersection with each set of F and it fulfills the condition to be a possible minimal hitting set. Now we have to check if this possible minimal hitting set is really minimal. It is definitely a hitting set but we are looking only for the minimal hitting sets. If a checked node already exists and it is a proper subset of this possible hitting set we have to add cross mark as the node because it is not a minimal hitting set, otherwise we add check mark as the node. Similarly we continue in this algorithm and we get two minimal hitting sets: $\{a, b\}$ and $\{b, c\}$.

Optimizations

Optimizations by Greiner et al. (1989) and Wotawa (2001) were added to Reiter's original algorithm a few years later.

Greiner claims that Reiter's way of creating a tree handles some situations incorrectly. The base algorithm is correct but prunning may lead to lose minimal hitting sets. Proposed way of preventing this lost is to use a directed acyclic graph instead of a tree. We did not use this optimization because in our case this situation can not happened. They provided a different approach for prunning. But the new prunning rule is applicable only then if input set F contains a strict superset of some other set from F. Our input does not contain such a superset.

The next optimization by Wotawa disclaims the proposed way by Greiner. He has returned to using a tree. He tries to save time and make the algorithm quicker. His idea is to create deterministic tree where nodes are sorted from left to right where left means the smallest size of an edge label and right means the biggest size of an edge label in one breadth level. In our case this optimization is also useless because in our tree the size of the edge label is always equal.

MergeXPlain

The main goal of this algorithm is to find some minimal conflicts. That is a complex task that can be done by this algorithm. But the result is a subset of all minimal conflicts. On the other hand the algorithm runs in seconds which is so much quicker compared to Minimal Hitting Set algorithm. This algorithm is proposed by (Shchekotykhin et al., 2015). The main algorithm is MergeXPlain that finds some minimal conflicts and it uses an algorithm that is called QuickXPlain. The QuickXPlain algorithm uses divide and conquere strategy and it returns one minimal conflict if such conflict exists.

To characterize a conflict we have to first characterize what is a diagnosable system and diagnosis from (Reiter, 1987).

Definition 2.4.1 ***Diagnosable System***

A diagnosable system is a pair (SD, COMPS) where SD is a system description (a set of logical sentences) and COMPS represents the system's components (a finite set of constants).

A diagnosis problem is when observations (OBS) are inconsistent with the diagnosable system.

Definition 2.4.2 ***Diagnosis***

Given a diagnosis problem (SD, COMPS, OBS), a diagnosis is a minimal set $\Delta \subseteq COMPS$ *such that* $SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg\{AB(c)|c \in COMPS \backslash \Delta\}$ *is consistent.*

A minimal conflict seems to be the same as minimal diagnosis. Finding all minimal conflicts corresponds to finding all minimal hitting sets (Reiter (1987)). Conflict CS is minimal if exists no proper subset of CS such as this subset is a conflict.

Definition 2.4.3 ***Conflict***

A conflict CS for (SD, COMPS, OBS) is a set $\{c_1, ..., c_k\} \subseteq COMPS$ *such that* $SD \cup OBS \cup \{\neg AB(c_i) | c_i \in CS\}$ *is incosistent.*

The QuickXPlain algorithm is designed for computation of explanations. In its current state it has the ability to return one conflict which means one possible explanation. It is a recursive algorithm showed in Algorithm 1.

Algorithm 1 QXP($\mathcal{B}$,$\mathcal{C}$)

Input: $\mathcal{B}$: Background theory, $\mathcal{C}$: the set of possibly faulty constraints

Output: A minimal conflict $CS \subseteq C$

if *isConsistent*$(\mathcal{B} \cup \mathcal{C})$ **then**
    return `"no conflict"`
else if $\mathcal{C} = \emptyset$ **then**
    return $\emptyset$
end if
return *GETCONFLICT*$(\mathcal{B}, \mathcal{B}, \mathcal{C})$

function GetConflict$(\mathcal{B}, D, \mathcal{C})$
    if $D \neq \emptyset \wedge \neg$*isConsistent*$(\mathcal{B})$ **then**
        return $\emptyset$
    end if
    if $|\mathcal{C}| = 1$ **then**
        return $\mathcal{C}$
    end if
    Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
    $D_2 \leftarrow$ GetConflict$(\mathcal{B} \cup \mathcal{C}_1, \mathcal{C}_1, \mathcal{C}_2)$
    $D_1 \leftarrow$ GetConflict$(\mathcal{B} \cup D_2, D_2, \mathcal{C}_1)$
    return $D_1 \cup D_2$
end function

At first the QuickXPlain algorithm is checking consistency of $\mathcal{B}$ and $\mathcal{C}$. There is no conflict if they are consistent so algorithm returns "no conflict". Otherwise the next check in the algorithm is if $\mathcal{C}$ is empty. If it is empty there can not be any conflict. Then algorithm calls method *GETCONFLICT* with parameters $\mathcal{B}$, $\mathcal{B}$ and $\mathcal{C}$. The *GETCONFLICT* method is checking if the second parameter D is not empty and if $\mathcal{B}$ is not consistent. Otherwise if $\mathcal{C}$ has size 1 then the result is $\mathbb{C}$ because $\mathcal{C}$ can not be divided into two sets anymore. It is already a conflict. If algorithm continues then $\mathcal{C}$ is randomly divided into two sets $\mathcal{C}_1$ and $\mathcal{C}_2$. Then the method *GETCONFLICT* is recursively called, first with parameters $\mathcal{B} \cup \mathcal{C}_1$, $\mathcal{C}_1$ and $\mathcal{C}_\in$, second with $\mathcal{B} \cup D_2$, D_2 and $\mathcal{C}_1$. After returning from recursion the results D_1 and D_2 are joined and returned as result.

Difference between QuickXPlain and MergeXPlain algorithm (Algorithm 2) is that MergeXPlain can return multiple conflicts at a time. MergeXPlain is repeatedly calling QuickXPlain for computation of an conflict.

Algorithm 2 MXP($\mathcal{B}$,$\mathcal{C}$)

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

if $\neg$*isConsistent*($\mathcal{B}$) **then**
  return `"no solution"`
else if *isConsistent*($\mathcal{B} \cup \mathcal{C}$) **then**
  return $\emptyset$
end if
$\langle \_, \Gamma \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}$)
return Γ

function FindConflicts($\mathcal{B}$,$\mathcal{C}$) **returns** a tuple $\langle \mathcal{C}', \Gamma \rangle$
  if *isConsistent*($\mathcal{B} \cup \mathcal{C}$) **then**
    return $\langle \mathcal{C}, \emptyset \rangle$
  else if $|\mathcal{C}| = 1$ **then**
    return $\langle \emptyset, \{\mathcal{C}\} \rangle$
  end if
  Split $\mathcal{C}$ into disjoint, non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$
  $\langle \mathcal{C}'_1, \Gamma_1 \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}_1$)
  $\langle \mathcal{C}'_2, \Gamma_2 \rangle \leftarrow$ FindConflicts($\mathcal{B}$,$\mathcal{C}_2$)
  $\Gamma \leftarrow \Gamma_1 \cup \Gamma_2$
  while $\neg$*isConsistent*($\mathcal{C}'_1 \cup \mathcal{C}'_2 \cup \mathcal{B}$) **do**
    $X \leftarrow$ GetConflict($\mathcal{B} \cup \mathcal{C}'_2, \mathcal{C}'_2, \mathcal{C}'_1$)
    $CS \leftarrow X \cup$ GetConflict($\mathcal{B} \cup X, X, \mathcal{C}'_2$)
    $\mathcal{C}'_1 \leftarrow \mathcal{C}'_1 \backslash \{a\}$ where $a \in X$
    $\Gamma \leftarrow \Gamma \cup \{CS\}$
  end while
  return $\langle \mathcal{C}'_1 \cup \mathcal{C}'_2, \Gamma \rangle$
end function

At the beginning of the MergeXPlain algorithm, the algorithm is checking consistency of $\mathcal{B}$. If $\mathcal{B}$ is already inconsistent it has no point to continue and algorithm returns no solution. If $\mathcal{B}$ and $\mathcal{C}$ are consistent together, no conflict can be found and algorithm returns empty set. Otherwise algorithm calls a method *FINDCONFLICTS* with parameters $\mathcal{B}$ and $\mathcal{C}$. This method checks consistency of $\mathcal{B}$ and $\mathcal{C}$ joined together. If they are consistent method returns a pair of $\mathcal{C}$ and empty set. If $\mathcal{C}$ has size one, method returns a pair of empty set and $\{\mathcal{C}\}$. This condition is here because if $\mathcal{C}$ has size one it can not be divided into two disjoint non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$. If $\mathcal{C}$ has size zero then the first condition is fulfilled because it is consistent with the set $\mathcal{B}$. But if the set $\mathcal{C}$ has size bigger than one then algorithm continues and splits the set $\mathcal{C}$ into two disjoint non-empty sets $\mathcal{C}_1$ and $\mathcal{C}_2$.

The method recursively calls itself, first with the set $\mathcal{C}_1$ as the second parameter and secondly with the set $\mathcal{C}_2$ as the second parameter. This is repeated until the method is emerged from both recursions and found explanations are joined together into variable Γ. The pair which is returned by function *FINDCONFLICTS* returns remaining elements of $\mathcal{C}$ as the first argument and set of explanations as the second argument. The method did not check the whole search space. It can happened that some part of explanation is in the set $\mathcal{C}_1$ and other part is in the set $\mathcal{C}_2$. To find at least some explanations there is a while loop that should find more explanations.

The following while loop is running until the set $\mathcal{C}_1'$ is either empty or consistent with the union of the sets $\mathcal{B}$ and $\mathcal{C}_2'$. In the while loop from the set $\mathcal{C}_1'$ are removed elements that is why this condition can be fulfilled. In the while loop the method calls the above mentioned method *GETCONFLICT* which returned one conflict if exists. This conflict is stored into variable X. Then the *GETCONFLICT* method is called again but with X as parameter and the result is stored into variable CS. It is also joined with the previous result X. Then comes the most important part of this loop.

From the set $\mathcal{C}_1'$ is removed α where α belongs to the set X. This can be interpreted differently. It could mean that always only one element is removed from the set $\mathcal{C}_1'$. Or it could mean that more elements are removed from this set. In the line below Γ represents all found explanations in this run of the method. After jumping from while loop the methods returns a pair or remaining literals $(\mathcal{C}_1' \cup \mathcal{C}_2')$ and found explanations. MergeXPlain algorithm is sound but is is not complete because it can not always find all explanations. This problems happens when they are explanations that are overlapping ecach other. Let's show the contradiction on following example.

Example 2.4.1 ***MergeXPlain example***

Let $\mathcal{K} = \{A \sqcap B \sqsubseteq D, A \sqcap C \sqsubseteq D\}$ *and let* $O = D(a)$*.*

Let us ignore negated ABox expressions and start with $Abd = \{A(a), B(a), C(a)\}$*. There are two minimal explanations of* $\mathcal{P} = (\mathcal{K}, O)$*:* $\{A(a), B(a)\}$*, and* $\{A(a), C(a)\}$*. Calling MXP(*$\mathcal{K} \cup \{\neg O\}, Abd$*), it passes the initial tests and calls* FINDCONFLICTS*(*$\mathcal{K} \cup \{\neg O\}, Abd$*).*

FINDCONFLICTS *needs to decide how to split* $\mathcal{C} = Abd$ *into* $\mathcal{C}_1$ *and* $\mathcal{C}_2$*. Let us assume the split was* $\mathcal{C}_1 = \{A(a)\}$ *and* $\mathcal{C}_2 = \{B(a), C(a)\}$*. Since both* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are now conflict-free w.r.t.* $\mathcal{K} \cup \{\neg O\}$*, the two consecutive recursive calls return* $\langle \mathcal{C}_1', \emptyset \rangle$ *and* $\langle \mathcal{C}_2', \emptyset \rangle$ *where* $\mathcal{C}_1' = \{A(a)\}$ *and* $\mathcal{C}_2' = \{B(a), C(a)\}$*.*

In the while loop, GETCONFLICT*(*$\mathcal{K} \cup \{\neg O\} \cup \{B(a), C(a)\}$*,* $\{B(a), C(a)\}$*,* $\{A(a)\}$*) returns* $X = \{A(a)\}$ *while* GETCONFLICT*(*$\mathcal{K} \cup \{\neg O\} \cup \{A(a)\}$*,* $\{A(a)\}$*,* $\{B(a), C(a)\}$*) returns* $B(a)$*, and hence the first confclit* $\gamma = \{A(b), B(a)\}$ *is found and added into* Γ*.*

However, consecutively $A(a)$ *is removed from* C_1' *leaving it empty, and thus the other conflict is not found and* $\Gamma = \{\{A(b), B(a)\}\}$ *is returned.*

Our approach

In this chapter we would like to introduce two ways of abductive reasoning which we have implemented. The first one is using Minimal Hitting Set (MHS) and the second one is MergeXPlain (MXP). The first one is based one Reiter's algorithm Minimal Hitting Set from (Reiter, 1987) but it is modified to include ABox abduction according to previous work from (Pukancová and Homola, 2016). The second one is based on MergeXPlain from (Shchekotykhin et al., 2015).

Our approach based on Minimal Hitting Set

Reiter's algorithm describes minmal hitting sets but we have to include also abductive reasoning as is described in algorithm 3. So our input data are knowledge base $\mathcal{K}$ and observation $\mathcal{O}$. Result from this algorithm are minimal explanations.

The MHS algorithm starts with creating a negation model $\neg\mathcal{M}$ which is calculated from $\mathcal{K}$ joined with the negation of $\mathcal{O}$. If there is no such model $\neg\mathcal{M}$ respectively $\neg\mathcal{M}$ is equal to null no explanations can be found. Otherwise HS-tree is created with root r. The node of root r is the found model $\neg\mathcal{M}$. Then the algorithm goes through each element of $\neg\mathcal{M}$ and creates a new child from root r. The new child is labeled on the edge by the current element. The algorithm generates HS-tree by breadth first search. So the next step is to iterate over each child in the next depth and to create its successors. But before generating its successors the algorithm must first check if the branch can be pruned or it is already an explanation. If it is already an explanation and its minimal it will be added to the result. To the result we add the path from root to the current node n which we can name as $H(n)$. If it is not minimal but it is an explanation we can prune it because it has no point to continue in this branch. If the branch contains a clash we can prune it too. If the branch is not pruned and it is not a minimal explanation then we have to calculate a negation model $\neg\mathcal{M}_i$ from $\mathcal{K}$ joined with $\neg\mathcal{O}$ joined with $H(n)$ similarly like in the first step when we generated model $\neg\mathcal{M}$. With this conditions we will get to the state that there is no such a branch that we can continue with and algorithm terminates. This algorithm has an exponential time complexity but it is sound and complete and it always terminates.

Algorithm 3 MHS($\mathcal{K}$,O)

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

```
M ← a model M of K ∪ {¬O}
if M = null then
    return "nothing to explain"
end if
create new HS-tree T = (V, E, L) with root r
label r by L(r) ← Abd(M)
for each σ ∈ L(r) create a successor n_σ of r and label the resp. edge by σ
S_E ← {}
while there is next node n in T w.r.t. BFS do
    if n can be pruned then
        prune n
    else if there is a model M of K ∪ {¬O} ∪ H(n) then
        label n by L(n) ← Abd(M)
    else
        S_E ← S_E ∪ {H(n)}
    end if
    for each σ ∈ L(n) create a successor n_σ of n and label the resp. edge by σ
end while
return S_E
```

Our approach based on MergeXPlain

Implementation

We use three reasoners, HermiT, JFact and Pellet. We use them only for consistency checking which means that in both algorithms (MHS, MXP) we need to ask the reasoner if our ontology is consistent in current state. We do not use all three reasoners at the same time but we use only one. The ontology is changing in the flow of algorithm, so we ask many times if ontology is consistent or other words if the ontology is clash free.

Evaluation

??? citovat, prepisat?

A preliminary experimental evaluation was conducted with implementations of MHS and MXP, both paired with three DL reasoners – Pellet, HermiT, and JFact. Both algorithms are implemented in Java and communicate with the reasoners through OWL API. The source code of both implementations is available online.[1]

The evaluation is split into two experiments. Experiment 1 is focused on computing explanations of size one. In this case MHS can be made more effective by bounding the HS-tree depth, and on the other hand MXP is complete in this case. Experiment 2 was conducted without any constraints on the size of explanations, but a timeout needed to be set. Both experiments were focused on comparing execution times between the two approaches and the three reasoners. Each time was computed as an average value from ten runs with ten different observations.

[1] https://github.com/katuskaa/MasterThesis

Dataset and Methodology

??? citovat, prepisat?

Three ontologies were chosen. The Family ontology[2], is our own ontology of family relations. It is smaller, but it is particularly useful in this use case as it generates a number of explanations of size higher than one. The second ontology, LUBM (Lehigh University Benchmark Guo et al. (2005)), is a standard benchmark. The Beer ontology[3] was chosen. Both LUBM and Beer were chosen because of their larger size compared to the Family ontology, but on the other hand as in the case of many real world ontologies their axiomatic structure is less complex which implies that most if not all explanations are of size one.

Table 1: Parameters of the ontologies

Ontology	**Concepts**	**Roles**	**Individuals**	**Axioms**
Family ontology	10	1	0	28
Beer ontology	58	9	0	165
LUBM	43	25	0	243

[2] http://dai.fmph.uniba.sk/~pukancova/aaa/ont/family2.owl
[3] https://www.cs.umd.edu/projects/plus/SHOE/onts/beer1.0.html

Experiment 1

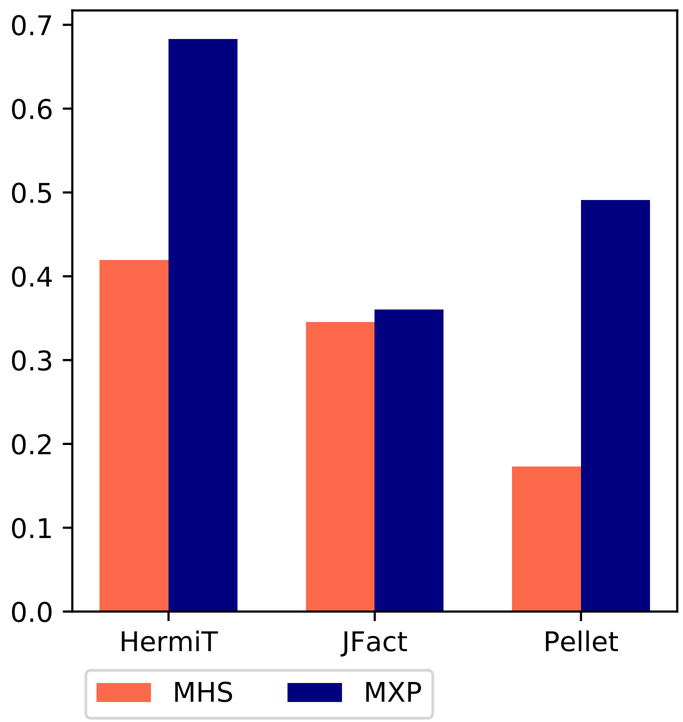

Figure 2: Family ontology

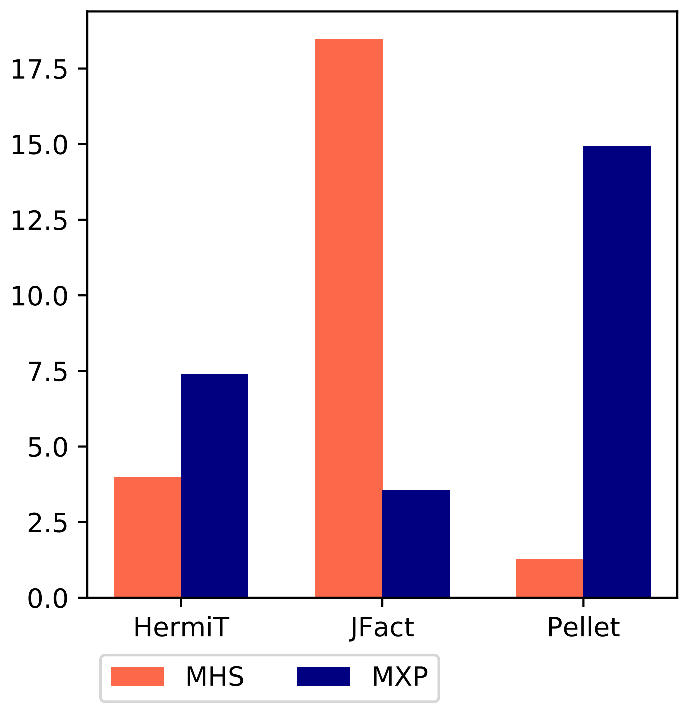

Figure 3: Beer ontology

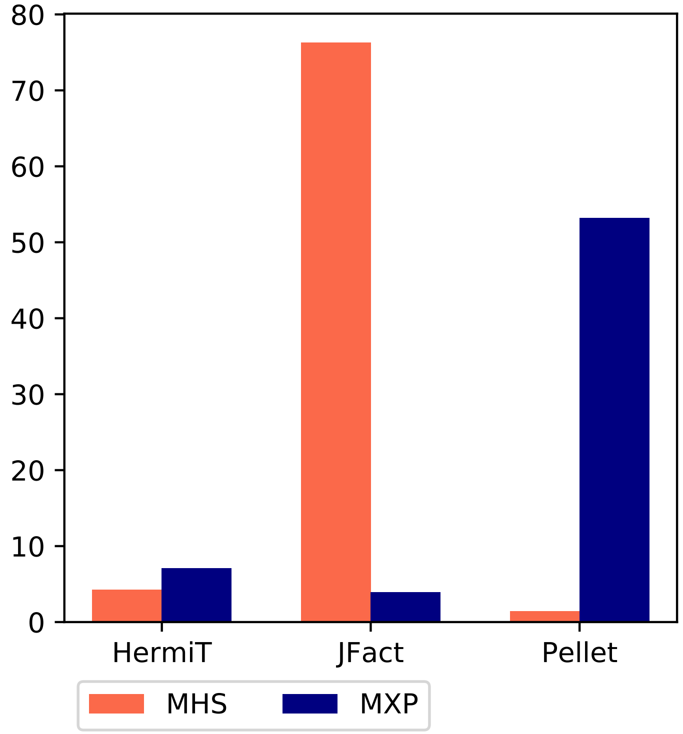

Figure 4: LUBM ontology

Experiment 2

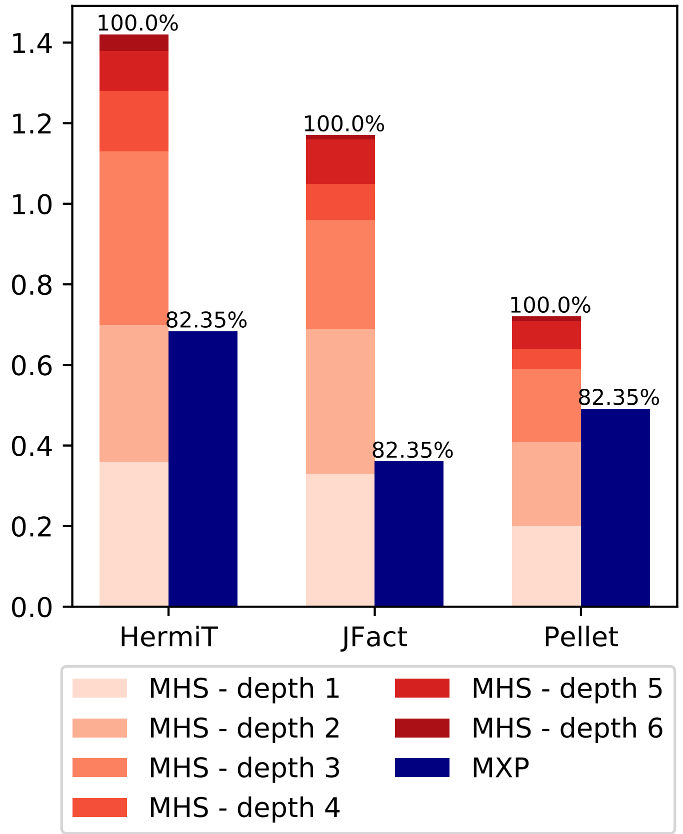

Figure 5: Family ontology

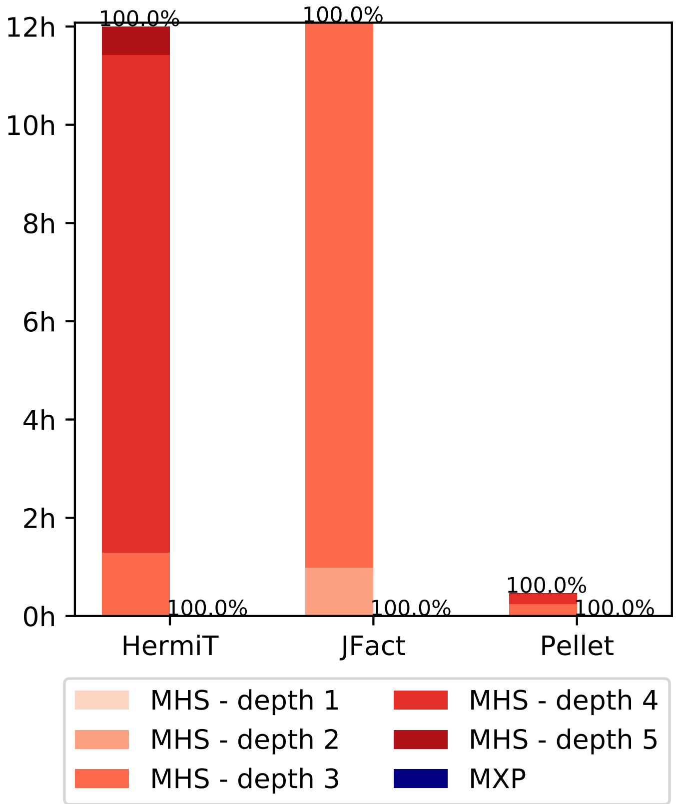

Figure 6: Beer ontology

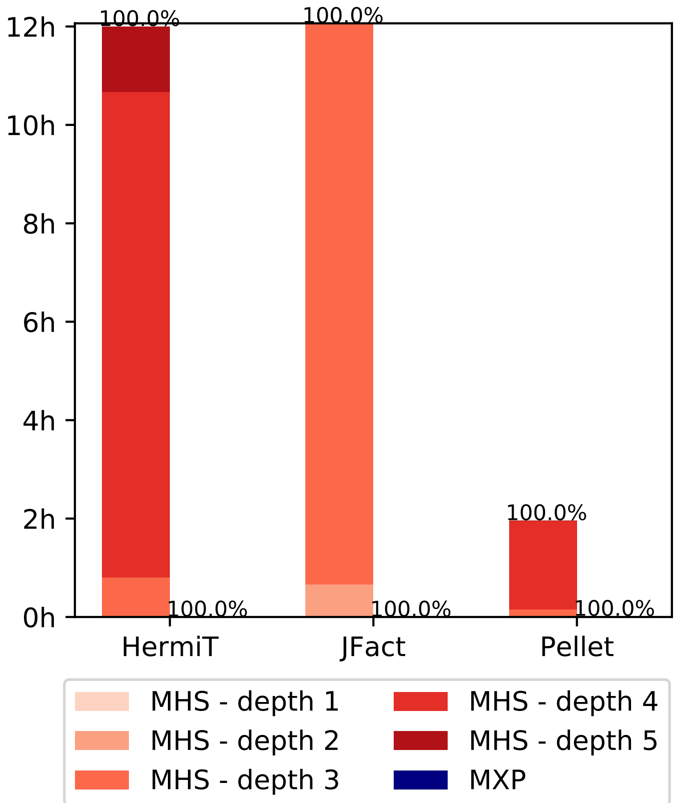

Figure 7: LUBM ontology

Conclusion from experiments

Conclusion

References

Sebastian Rudolph. Foundations of description logics. In *Reasoning Web. Semantic Technologies for the Web of Data - 7th International Summer School 2011, Galway, Ireland, August 23-27, 2011, Tutorial Lectures*, pages 76–136, 2011. doi: 10.1007/978-3-642-23032-5_2. URL https://doi.org/10.1007/978-3-642-23032-5_2.

Stuart James Fitz-Gerald and Bob Wiggins. Staab, s., studer, r. (eds.), handbook on ontologies, series: International handbooks on information systems, second ed., vol. XIX (2009), 811 p., 121 illus., hardcover £164, ISBN: 978-3-540-70999-2. *Int J. Information Management*, 30(1):98–100, 2010. doi: 10.1016/j.ijinfomgt.2009.11.012. URL https://doi.org/10.1016/j.ijinfomgt.2009.11.012.

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

Júlia Pukancová and Martin Homola. Tableau-based abox abduction for the ALCHO description logic. In *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017.*, 2017. URL http://ceur-ws.org/Vol-1879/paper11.pdf.

Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. doi: 10.1016/0004-3702(87)90062-2. URL https://doi.org/10.1016/0004-3702(87)90062-2.

Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in reiter's theory of diagnosis. *Artif. Intell.*, 41(1):79–88, 1989. doi: 10.1016/0004-3702(89)90079-9. URL https://doi.org/10.1016/0004-3702(89)90079-9.

Franz Wotawa. A variant of reiter's hitting-set algorithm. *Inf. Process. Lett.*, 79(1):45–51, 2001. doi: 10.1016/S0020-0190(00)00166-6. URL https://doi.org/10.1016/S0020-0190(00)00166-6.

Kostyantyn M. Shchekotykhin, Dietmar Jannach, and Thomas Schmitz. MergeXplain: Fast computation of multiple conflicts for diagnosis. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina*. AAAI Press, 2015.

Júlia Pukancová and Martin Homola. Tableau-based abox abduction for description logics: Preliminary report. In *Proceedings of the 29th International Workshop on Description Logics, Cape Town, South Africa, April 22-25, 2016.*, 2016. URL http://ceur-ws.org/Vol-1577/paper_23.pdf.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3(2-3):158–182, 2005.

Appendices