Chapter 2

Background and Related Work

Knowledge representation (KR) is an area of artificial intelligence research which deals with representing knowledge using formal symbols, thus allowing the application of automated reasoning techniques to infer new knowledge from given knowledge [BL04]. KR forms the basis of *knowledge-based systems*, 'intelligent' systems which make use of a *knowledge base* (KB) that contains told facts about some domain, as well as procedures to reason over these facts and infer implicit knowledge. While there exists a wide range of approaches to implementing knowledge-based systems, such as logic programming [BG94], frames [Min74], and semantic networks [Sow87], in this thesis we are dealing with description logics (DLs) [BCM+03] as the underlying knowledge representation formalism for knowledge bases. DLs are a family of logics based on a 2-variable fragment of first-order logic which is more expressive than propositional logic while still being decidable.

This chapter introduces the basic concepts of description logics, which underpin the Web Ontology Language OWL. It outlines the syntax and semantics of description logics and fixes relevant notions such as *axioms* and *entailments* of an OWL ontology. It also discusses the landscape of logical and non-logical errors occurring in OWL ontologies, which motivates the need for tailored debugging support. It then introduces *justifications* as an explanation service for entailments of OWL ontologies, and reviews the literature dealing with justification-based explanation. This covers approaches to computing single and multiple justifications, as well as the issues of understanding justifications, justification-based repair of errors, and coping with multiple justifications, which is the main focus of this thesis.

2.1 Description Logic Knowledge Bases

2.1.1 DL Syntax and Semantics

DL Syntax

The main building blocks of DLs are atomic *concepts*, *roles* and *individuals*. With respect to their relationship with FOL, concepts correspond to unary predicates in FOL, roles to binary predicates and individuals to constants. These entities are used to create more expressive concept expressions with the help of constructors, whereby the available constructors depend on the expressivity of the respective DL. As a convention, we shall be using the upper-case letters A and B for atomic concepts and C, D, ... for possibly complex concepts; the lower-case letters r, s, ... for role names, and the lower-case letters a, b, ... for individuals.

The basic description logic $\mathcal{ALC}$ ('Attribute Logic with Complement'), which many other more expressive DLs build upon, allows the constructors given in Table 2.1, where C and D denote concepts and r is a role name.

Constructor	Syntax
Top concept	$\top$
Bottom concept	$\bot$
Concept negation	$\neg C$
Concept intersection (conjunction)	$C \sqcap D$
Concept union (disjunction)	$C \sqcup D$
Existential restriction	$\exists R.C$
Universal restriction	$\forall R.C$

Table 2.1: $\mathcal{ALC}$ constructors

The name of a description logic is generally comprised of mnemonics representing the available constructors in the respective logic: The letters $\mathcal{N}$ and $\mathcal{Q}$ stand for unqualified ($\geq nr$, $\leq nr$) and qualified number restrictions ($\geq nr.C$, $\leq nr.C$) , respectively, $\mathcal{F}$ represents the functionality of roles ($\geq nr$), $\mathcal{H}$ stands for role hierarchies ($r \sqsubseteq s$), $\mathcal{I}$ is the role inverse, $\mathcal{R}$ stands for complex role inclusions of the type $r \circ s \sqsubseteq r$. The letter $\mathcal{O}$ denotes the presence of nominals, which allows the use of individuals in the place of concepts in TBox axioms: $C \sqsubseteq \{a\}$. Some notable description logics include $\mathcal{S}$ which corresponds to $\mathcal{ALC}+$ ($\mathcal{ALC}$ plus role transitivity), $\mathcal{EL}$ which allows existential quantifiers and intersection, $\mathcal{EL}^{++}$ which corresponds to $\mathcal{EL}$ plus complex role inclusions and nominals, and the highly expressive logics $\mathcal{SHOIN}(D)$ and $\mathcal{SROIQ}(D)$ which underpin OWL and OWL 2, respectively. In the context of OWL, the suffix (D) indicates the use of XML Schema[1] datatypes.

[1] `http://www.w3.org/TR/xmlschema11-2/`

Axioms

A description logic knowledge base $\mathcal{K}$ is generally regarded as a set of axioms which are *asserted* in the KB. Axioms are sentences that make statements about the domain knowledge modelled in the KB. The axioms in a KB are classified into the sets of *TBox*, *RBox* and *ABox* axioms which are denoted as $\mathcal{T}$, $\mathcal{R}$ and $\mathcal{A}$, respectively: $\mathcal{K} = \langle \mathcal{T}, \mathcal{R}, \mathcal{A} \rangle$. The *signature* $sig(\mathcal{K})$ of a knowledge base $\mathcal{K}$ is the set of all concept, role, and individual names occurring in $\mathcal{K}$.

A TBox axiom α is either a *subsumption* ($C \sqsubseteq D$) or *equivalence* ($C \equiv D$) between two (possibly complex) concepts C and D in an ontology. A subsumption axiom expresses that C is a sub-concept of D, that is, every instance of C is also an instance of D. We can say that C 'is-a' D. Equivalence axioms state that two concepts C and D are equivalent, which corresponds to a bi-directional subsumption $C \sqsubseteq D$ and $D \sqsubseteq C$.

Subsumption and equivalence are also possible between roles, which are described by RBox axioms: $r \sqsubseteq s$ specifies that r is a sub-role of s, which means that every two individuals that have an r-relationship also have an s-relationship between them. $r \equiv s$ expresses that the two roles are equivalent.

ABox axioms make statements about the relations between individuals and concepts, and between individuals and roles: $C(a)$ expresses that the individual a is an instance of the concept C, and $r(a, b)$ specifies that there exists an r-relationship between the individuals a and b.

As an example, we use a small knowledge base that describes the eating habits of animals on an abstract level:

Example 2.

$$
\begin{aligned}
\mathcal{K} = \{ & \mathsf{Cat} \sqsubseteq \mathsf{Carnivore} \\
& \mathsf{Carnivore} \sqsubseteq \mathsf{Animal} \sqcap \forall \mathsf{eats.Animal} \\
& \mathsf{Plant} \sqsubseteq \neg \mathsf{Animal} \\
& \mathsf{Grass} \sqsubseteq \mathsf{Plant} \\
& \mathsf{PetOwner} \sqsubseteq \mathsf{Human} \sqcap \exists \mathsf{hasPet.Animal} \\
& \mathsf{Cat}(\mathsf{Molly}) \\
& \mathsf{Human}(\mathsf{Alice}) \\
& \mathsf{hasPet}(\mathsf{Alice}, \mathsf{Molly}) \}
\end{aligned}
$$

The TBox of this example KB consists of the first five axioms, which make statements about the concepts in the domain: Cats are carnivores, a carnivore is an animal

$$
\begin{aligned}
A^{\mathcal{I}} &\subseteq \triangle^{\mathcal{I}} \\
r^{\mathcal{I}} &\subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}} \\
\top^{\mathcal{I}} &:= \triangle^{\mathcal{I}} \\
\bot^{\mathcal{I}} &:= \emptyset \\
(\neg C)^{\mathcal{I}} &:= \triangle^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists r.C)^{\mathcal{I}} &:= \{x \mid \exists y.^{\mathcal{I}} \langle x, y\rangle \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \\
(\forall r.C)^{\mathcal{I}} &:= \{x \mid \forall y.^{\mathcal{I}} \langle x, y\rangle \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}
\end{aligned}
$$

Table 2.2: Interpretation function for the description logic $\mathcal{ALC}$

which only eats animals, plants are disjoint with animal (i.e. no one thing can be a plant and an animal at the same time), grass is a plant, and a pet owner is a human who has an animal as a pet. The last three axioms make up the ABox, which contains statements about individuals (Molly is a Pet, Alice is a Human), and the relationship between them (Molly is Alice's pet).

Axioms containing only atomic concepts on the left-hand side are called *definitions*, and an *acyclic* TBox containing only these types of axioms where all concepts on the left-hand side have unique names is called a *definitorial* TBox. *General* TBoxes may contain *general concept inclusion* (GCI) axioms, which allow complex concept expressions on both the right- and the left-hand side: $\exists \mathsf{eats}.\top \sqsubseteq \mathsf{Animal}$

Model-Theoretic Semantics

The semantics of description logics is model-theoretic and given by interpretations. An interpretation $\mathcal{I}$ is a tuple $\langle \triangle^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\triangle^{\mathcal{I}}$ is the interpretation domain, that is, a non-empty set of elements, and $\cdot^{\mathcal{I}}$ the interpretation function. This mapping function maps concept names from the ontology to subsets of $\triangle^{\mathcal{I}}$, role names to subsets of $\triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$ and individuals to elements in $\triangle^{\mathcal{I}}$. The interpretation function for concepts, roles and individuals in $\mathcal{ALC}$ is defined in Table 2.2

An interpretation $\mathcal{I}$ is a *model* for an axiom α if it satisfies the axiom, which is expressed as $\mathcal{I} \models \alpha$. A model of a knowledge base $\mathcal{K}$ is an interpretation in which all axioms in $\mathcal{K}$ are satisfied: $\mathcal{I} \models \mathcal{K}$.

2.1.2 Standard Reasoning Services

The ability to convey information without having to explicitly state it is one of the main advantages of logic-based knowledge bases. Description logic *reasoners* are pieces of software which employ efficient algorithms to provide reasoning services over a given input knowledge base $\mathcal{K}$:

Consistency Given a knowledge base $\mathcal{K}$, determine whether there exists an interpretation $\mathcal{I}$ for $\mathcal{K}$ such that $\mathcal{I} \models \mathcal{K}$. If there exists such an interpretation, return 'true' (the KB is consistent), otherwise return 'false' (the KB is inconsistent). All other reasoning services can be reduced to consistency checking.

Satisfiability A concept C is satisfiable if it is not necessarily empty in *some* model of the ontology: $C^{\mathcal{I}} \neq \emptyset$ in some $\mathcal{I}$ that is a model of the ontology. C is unsatisfiable, denoted as $C \sqsubseteq \bot$, if it is mapped to the empty set in *all* models of the ontology: $C^{\mathcal{I}} = \emptyset$

Subsumption One of the main reasoning task in DLs is *subsumption* between concept expressions, i.e. checking whether a (potentially complex) concept in the ontology is a subset of another set. The subsumption between two concepts where C is necessarily interpreted as a subset of D is denoted $C \sqsubseteq D$. This means that each domain element that is in the set given by the interpretation of C is also in the interpretation of D: $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Equivalence Two concepts C and D are equivalent in the ontology ($C \equiv D$), if it holds that all elements of $C^{\mathcal{I}}$ are in the set $D^{\mathcal{I}}$ and vice versa in all models of the ontology, i.e. if $C^{\mathcal{I}} = D^{\mathcal{I}}$. Equivalence is a special case of subsumption.

Instantiation Instantiation (or instance checking) checks which individuals are instances of a particular concept: Given an individual a, a concept C and a KB $\mathcal{K}$, return 'true' if $a^{\mathcal{I}} \in C^{\mathcal{I}}$ in all models of $\mathcal{K}$.

There exist a number of sound and complete algorithms for providing the above reasoning services, which are implemented by description logic reasoners: *Tableaux* [DL96, Hor97, HST00, BS00] procedures attempt to build a finite tree-like model of each concept in the ontology using *tableau rules*. The concepts are first translated into negation normal form (NNF) and then decomposed according to the respective tableau rules. The nodes and edges in the tree are labelled with the decomposed concepts and roles, respectively. A *clash* in the tree occurs when the algorithm attempts to label a node in the tableaux tree with contradictory concepts, such as an atomic concept and its negation.

When no more rules are applicable to the concept fragments, or when a clash occurs, the algorithm terminates. If some clash-free tree (i.e. some model) can be found for a concept C, the concept is satisfiable. Likewise, if no clash-free tree can be found, the concept has no models, i.e. it is unsatisfiable. The types of and usage of constructors in

a description logic affect the types of rules used in a tableau algorithm, which in turn can increase the computational complexity of the reasoning algorithm. For instance, as some of the tableau rules are non-deterministic (i.e. involve choice), the algorithm has to perform backtracking if such a rule has been applied. There exists a number of highly sophisticated optimisations for tableaux algorithms that ensure early branch determination and clash detection, which improve the performance of the algorithms significantly.

Entailments

An *entailment* η of a knowledge base $\mathcal{K}$ is an axiom that follows logically from $\mathcal{K}$. $\mathcal{K}$ entails η ($\mathcal{K} \models \eta$) if η is true in *all* models $\mathcal{I}$ of $\mathcal{K}$. The entailment relation is *monotonic* in DLs, which means that entailments are preserved when further axioms are added to the KB. Every axiom which is asserted in $\mathcal{K}$ is an entailment of the knowledge base, while there are other entailments which represent *implicit* knowledge in the KB. For example, the pet KB above explicitly asserts that $\mathsf{Cat} \sqsubseteq \mathsf{Carnivore}$ and $\mathsf{Carnivore} \sqsubseteq \mathsf{Animal} \sqcap \forall \mathsf{eats}.\mathsf{Animal}$. From these statements we can conclude that every Cat is an Animal: $\mathcal{K} \models \mathsf{Cat} \sqsubseteq \mathsf{Animal}$. There exist practical decision procedures to make this information explicit; description logic *reasoners* and the standard reasoning tasks are discussed in the following section.

Entailments are not restricted to any specific axiom type; for example, the axioms $\mathsf{Cat} \sqsubseteq \mathsf{Carnivore}$ and $\mathsf{Carnivore} \sqsubseteq \mathsf{Animal} \sqcap \forall \mathsf{eats}.\mathsf{Animal}$ also entail the statement $\mathsf{Cat} \sqsubseteq \forall \mathsf{eats}.\mathsf{Animal}$. The set of all axioms which are valid in a description logic $\mathcal{L}$ and entailed by a KB $\mathcal{K}$ is called the *deductive closure* of $\mathcal{K}$:

Definition 1 (Deductive closure)**.** *The deductive closure $\mathcal{K}^*_{\mathcal{L}}$ of a knowledge base $\mathcal{K}$ is the set of axioms in the description logic $\mathcal{L}$ such that $\mathcal{K}^*_{\mathcal{L}} = \{\alpha \in \mathcal{L} \mid \mathcal{O} \models \alpha\}$. When clear from the context, the subscript $\mathcal{L}$ is dropped.*

While even KBs based on weakly expressive[2] DLs have infinitely many entailments (i.e. the deductive closure is infinite), the focus when constructing and analysing a knowledge base often lies on the *concept hierarchy* of the KB, which is represented by its entailed atomic subsumption axioms. In the context of this thesis, we simply speak of an *entailment set* to denote a *finite* set of entailments of a knowledge base $\mathcal{K}$, restricted by some clearly specified criteria:

Definition 2 (Entailment set)**.** *An entailment set $\varepsilon_{\mathcal{K}}$ of a knowledge base $\mathcal{K}$ is a finite set of axioms $\{\alpha_i \mid i \in \mathbb{N}\} \subseteq \mathcal{K}^*$.*

[2] As even the most basic description logics, $\mathcal{AL}$, $\mathcal{EL}$, and $\mathcal{FL}$ allow concept intersection, which can be used to generate an infinite number of—tautological—entailments.

An in-depth discussion of the issue of specifying useful entailment sets follows in Chapter 3.

Incoherence

A concept is *unsatisfiable* if it is mapped to the empty set in its interpretation, which is denoted as $C^{\mathcal{I}} = \emptyset$. This means that there exists no model where C is non-empty, i.e. the concept cannot have any instances. This is caused by contradictory statements, such as $C \sqsubseteq D \sqcap \neg D$. A KB that contains some unsatisfiable concept is called *incoherent*.

Continuing with the previous pet KB, we can induce incoherence in the KB by adding the last TBox axiom which causes the concept $\mathsf{SickCat}$ to be entailed as unsatisfiable:

Example 3 (Incoherent Pet Knowledge Base)**.**

$$
\begin{aligned}
\mathcal{K}' = \{&\mathsf{Cat} \sqsubseteq \mathsf{Carnivore}\\
&\mathsf{Carnivore} \sqsubseteq \mathsf{Animal} \sqcap \forall \mathsf{eats}.\mathsf{Animal}\\
&\mathsf{Plant} \sqsubseteq \neg \mathsf{Animal}\\
&\mathsf{Grass} \sqsubseteq \mathsf{Plant}\\
&\mathsf{PetOwner} \equiv \mathsf{Human} \sqcap \exists \mathsf{hasPet}.\mathsf{Animal}\\
&\mathsf{SickCat} \equiv \mathsf{Cat} \sqcap \exists \mathsf{eats}.\mathsf{Grass}\\
&\mathsf{Cat}(\mathsf{Molly})\\
&\mathsf{Human}(\mathsf{Alice})\\
&\mathsf{hasPet}(\mathsf{Alice}, \mathsf{Molly})\}
\end{aligned}
$$

The conflict is caused by the TBox axioms in $\mathcal{K}$' which entail that cats are carnivores, thus eating only animals, but that sick cats eat grass, a concept which is a subconcept of plant, thus known to be disjoint with animal. As, according to our knowledge base, there cannot exist a carnivore who also eats grass, the concept $\mathsf{SickCat}$ cannot have any elements in its interpretation: $SickCat^{\mathcal{I}} = \emptyset$. Therefore, the concept $\mathsf{SickCat}$ is said to be unsatisfiable, and the knowledge base $\mathcal{K}$' is incoherent.

Inconsistency

While incoherence itself does not cause any reasoner problems (as the concept simply correlates to the empty set in all models), instantiation of unsatisfiable concepts causes the KB to be contradictory. If the axiom $\mathsf{SickCat}(\mathsf{Molly})$ is added to the KB in which $\mathsf{SickCat}$ is unsatisfiable, it is not possible for the KB to have any model in which this

statement is satisfied. Such a knowledge base that has no models is called *inconsistent*, which is denoted as $\mathcal{K} \models \top \sqsubseteq \bot$.

An inconsistent KB has the property that it entails *everything* that can be expressed in the respective logic $\mathcal{L}$—since it is contradictory, it cannot possibly make any meaningful statements about the knowledge it models. Different approaches to reasoning with inconsistent knowledge bases have been explored, such as *paraconsistent* reasoning [MHL07], or the selection of a consistent subset based on a relevance function [HvT05].

2.1.3 The Web Ontology Language OWL

From Description Logic to OWL

While 'ontology' is a term borrowed from philosophy, in computer science it describes a software artefact representing information about the entities in a domain, and the relationships between them [Gru93]. In the remainder of this thesis, we will use the term *ontology* (denoted by the letter $\mathcal{O}$) to refer to a description logic knowledge base which is represented in some machine-processable format, such as OWL.

OWL is a successor of the web ontology language DAML+OIL [Hor02], a description logic based ontology language with an RDF/XML syntax which evolved from merging DAML-ONT (a language developed by the DARPA Agent Markup Language programme) and OIL (Ontology Inference Layer, developed by the European *On-To-Knowledge* project). The first version of OWL, which is based on the expressive description logic $\mathcal{SHOIN}(D)$ and was described as a 'revision' of DAML+OIL, became an official W3C recommendation in February 2004.[3]

OWL 2, the successor of OWL, was made a W3C recommendation in 2009. It comprises two species of different expressivities, namely OWL 2 DL and OWL 2 Full. The underlying formalism of OWL 2 DL is the description logic $\mathcal{SROIQ}$ [HKS06]. This highly expressive DL adds a range of complex constructors and axiom types to those of $\mathcal{ALC}$, such as complex role inclusions (represented by the letter $\mathcal{R}$), nominals ($\mathcal{O}$), inverse roles ($\mathcal{I}$), and qualified number restrictions ($\mathcal{Q}$). In the context of this thesis we will simply regard OWL 2 DL as a syntactic variant of $\mathcal{SROIQ}$, using the name OWL in place of OWL 2 DL, unless otherwise stated.

Usage of OWL

OWL has been designed with the aim of providing an expressive machine-processable ontology language for use in various applications and domains. There exists a wide array of tools and libraries for creating and editing OWL ontologies, such as Protégé 4,

[3] `http://www.w3.org/TR/owl-features/`

Top Braid Composer, Swoop, the OWL API,[4] a Java library that gives access to a large number of ontology editing tasks and reasoner interfaces, a number of highly optimised OWL reasoners, such as Pellet [SPC+07], HermiT [SMH08], and FaCT++ [TH06], as well as reasoners which are tuned towards specific subsets of OWL [KKS11]. Further, there are several syntaxes for OWL, such as the human-oriented Manchester Syntax [HDG+06], various RDF formats, and the OWL/XML serialisation. This broad set of tools allows users to choose a suitable development environment for their respective application requirements, and improved tool support is being regarded as one of the main reasons for the growing use of OWL ontologies as a knowledge representation mechanism [HPS08].

OWL 2 Profiles There exist three named 'profiles' for OWL 2, syntactic subsets of OWL 2 DL that are tailored towards different applications, trading expressivity of the language for efficient reasoning:

The OWL 2 EL profile is a tractable fragment of OWL 2 which is based on the description logic $\mathcal{EL}^{++}$. OWL 2 EL omits some of the more expressive OWL 2 constructors in favour of efficient reasoning (polynomial time), which makes it attractive for use in ontologies that 'contain very large numbers of properties and/or classes' [owl09].

OWL 2 QL (Query Language), which is based on the *DL-Lite* family of description logics [ACKZ09], has been defined with an eye toward applications which focus on query answering over large amounts of instance data. The queries executed against OWL 2 QL ontologies can be rewritten into SQL queries, which allows storing instance data in a standard relational database, thus significantly improving query performance. OWL 2 QL is the profile used for ontologies in OBDA (Ontology-Based Data Access) systems, which combine OWL 2 QL reasoning with efficient querying over relational databases [CGL+11, RMC12].

Reasoning systems for ontologies in the OWL 2 RL (Rule Language) profile can be implemented using rule-based reasoning engines. The profile restricts the use of constructors to certain positions in axioms, e.g. universal restrictions and negation are only allowed on the RHS of axioms. While OWL 2 RL allows the use of a wider range of expressive constructors than the other two OWL 2 profiles, these positional restrictions lead to efficient reasoning performance.

Semantic Web The idea of a Semantic Web [BLHL01] marks a move away from a human-oriented 'web of documents' towards a machine-oriented 'web of data'. By enriching web pages with semantic content we can express relationships between entities on the web, thus allowing understanding of those relationships by automated processes

[4] `http://owlapi.sourceforge.net/`

rather than direct human processing. This 'meaningful' interlinking of web content is thought to improve searching for and integrating information on the web.

As a 'source of shared, precisely defined terms' [Hor02], ontologies play a key role in the Semantic Web, with the majority of ontologies found on the web using OWL or RDF(S) [Car07]. Examples of prominent OWL ontologies on the web include the BBC Programmes and Wildlife ontologies[5], and the set of geographical ontologies created by Ordnance Survey.[6] However, while OWL has been designed specifically as an ontology language for the Semantic Web, its perceived complexity and unpredictable reasoning performance still prohibit wider uptake by web application developers, for whom 'a little semantics' in the form of RDF may often seem sufficient.

Medical Informatics and Life Sciences OWL ontologies are frequently used for the encoding of biological and medical knowledge as part of larger information systems. Examples for intensively used and maintained medical OWL ontologies include the SNOMED CT ontology[7] which contains medical terminology used in electronic health records, the OWL version of the International Classification of Diseases (ICD-10) catalogue[8] released by the World Health Organization (WHO), and the National Cancer Institute (NCI) thesaurus[9] which covers knowledge related to cancer, such as anatomy, findings, drugs, and genes. There exists a vast array of biological ontologies of various expressivities, such as the Gene Ontology, which aims to 'standardize the representation of gene and gene product attributes'. the Experimental Factor Ontology (EFO) which models variables in biological experiments, and various ontologies describing the anatomy of species.

2.2 Errors in OWL Ontologies

As OWL is a highly expressive ontology language, offering users a wide range of constructors for modelling domain knowledge at a high level of precision. The downside of this expressivity is that constructors can frequently be misinterpreted by users, and combinations of otherwise correct axioms and constructors may lead to undesired side-effects. Beyond side-effects caused by human users, common engineering tasks such as translation of an ontology into OWL from some other formalism, automated generation of an ontology using some input source, or integration of an existing ontology, may also introduce errors into an ontology.

[5] http://www.bbc.co.uk/ontologies/
[6] http://www.ordnancesurvey.co.uk/oswebsite/ontology/
[7] http://www.ihtsdo.org/snomed-ct/
[8] http://www.who.int/classifications/icd/en/
[9] http://ncit.nci.nih.gov/

We distinguish between different types of errors: *Logical errors*, which can be detected through the use of a reasoner, *factual errors*, which are incorrect statements with respect to the domain knowledge modelled in the ontology, and *modelling inconsistency*, which describes incorrect information with respect to the ontological commitment made in the ontology.

2.2.1 Logical Errors

Incoherence A common logical error is the *unsatisfiability* of a class A in $\mathcal{O}$, which implies that A cannot have any instances; an ontology containing some unsatisfiable class A is called *incoherent*. While the unsatisfiability of a class may be intentional (i.e. to explicitly prohibit a certain definition for a class), it mostly indicates a modelling error caused by contradictory statements. A large number of ontologies used in practice is indeed incoherent, which can give rise to another, more severe, logical error, namely the inconsistency of the ontology.

Inconsistency By instantiating an unsatisfiable class, i.e. specifying that an individual is a member of an unsatisfiable class, an ontology becomes *inconsistent*, or *contradictory*. Inconsistency is a severe logical error, as this makes it impossible to infer meaningful information from the ontology; this renders the ontology useless. In order to prevent the inconsistency of an ontology, it is highly desirable to *repair* any unsatisfiable classes as soon as they arise in the ontology engineering process.

Having said that, an unsatisfiable class does not necessarily imply an accidental logical error. On the contrary, it might have been introduced into the ontology for testing purposes. For example, the Protégé 4 OWL tutorial [Hor11b] suggests generating a subclass of two disjoint classes in order to see the effects of the disjointness axioms; in this case, the unsatisfiability of the newly created class is intentional and correct. While such a use of unsatisfiability may be a rare case, we have to bear this in mind when providing debugging support to ontology developers.

Wrong Entailments Some logical errors, such as incoherence and inconsistency, can be easily[10] detected by a reasoner; factual errors (wrong and unwanted entailments), however, can only be spotted by a user with the appropriate domain knowledge.

While there may be some knowledge in the world which we can consider facts (such as 'the parent of every human is a human'—for the time being, there is little to argue about this statement), we often come across knowledge and relationships which are and *cannot* be clearly defined. Thus, we can say that every ontology is subject to

[10] 'Easy' with respect to the time a reasoner requires for a consistency check of the ontology. Depending on the size and complexity of the ontology, this can be a rather demanding task.

an *ontological commitment* which prescribes relations in the ontology depending on the *view* we have on a domain. Going back to the parent–human relationship, what happens if we build a knowledge base that includes information about, say, fantasy creatures? In such an ontology we can imagine the need to allow a human being to have a non-human parent; yet, given our restriction that humans have only human parents, this would be impossible.

While fantasy creatures may be an unusual example, we can quickly see cases in which such a restriction can lead to more fundamental debates. A frequently occurring issue is that of gender: General knowledge tells us that there are two genders, and every person is either male or female, which we may model as such in an ontology. And yet, this is clearly an issue which ranges far beyond knowledge modelling techniques, as the gender binary has long been the focus of fundamental biological and political debate. When talking about *errors* in an ontology, we have to bear in mind that these errors may not be statements which are intrinsically wrong, but simply *wrong in the context of the ontology.*

Non-Entailments Missing entailments, or non-entailments, are frequently occurring errors in OWL ontologies which are particularly hard to detect. When constructing an ontology, developers expect certain entailments, such as an obvious subsumption between two classes, to be caused by the axioms they add; if an expected (and desired) entailment does not follow from the ontology, this may be considered an error. Roussey et al [RCVB09] found that ontology developers frequently add axioms to ontology that either do not have the desired effect, or no effect at all. In these cases, *ontology diff* tools (e.g. [FMV10, GPS11, KŠK11]) can help users spot whether any modifications to an ontology had the desired effect; beyond the detection of ineffectual modifications, however, dealing with non-entailments is a non-trivial task:

Typically, as a result of under-constraining, the number of entailments of an ontology is significantly smaller than the number of non-entailments; therefore, presenting all possible entailments (even if restricted to a reasonable finite set) to a user can quickly lead to a large and unmanageable set of axioms. A more structured approach to *ontology completion* is based on methods from the area of Formal Concept Analysis (FCA) [BGSS07]. An ontology completion tool presents users with a series of potential entailments, e.g. a subsumption between two classes, asking them to accept or reject the presented entailment. In case of an accepted entailment, an axiom is added to the ontology to explicitly state the relationship; otherwise, the tool adds a *counter-example* to ensure that the axiom is not entailed.

2.2.2 Non-Logical Errors

Structural Irregularities We have a good understanding of the common errors that users make when constructing OWL ontologies [RDH+04, KPSC06, RCVB09]. Some of these errors include confusing 'and' and 'or', subsumptions and equivalences, 'forgetting' previously defined property restrictions, and introducing axioms which do not have any effect on the model of the ontology. Given a catalogue of such *antipatterns*, an OWL tool can point out axioms that are potentially erroneous and provide further explanation of the semantics to a user. Kalyanpur et al [KPSC06] use a set of common antipatterns in a heuristics-based repair tool, which suggests removal or modification of axioms based on, amongst others, the likelihood of it them being incorrect.

Structural Irregularities As described above, there exists a number of ontology modelling patterns which, when chosen for an ontology, need to be adhered to. But even without formal patterns, ontologies often exhibit syntactic regularities due to habits or training of ontology engineers [MMIS12]. Regularities include 'good practice' strategies such as adding domain, range, and inverse for every newly introduced object property. Therefore, deviating from the given structure and existing regularities may be considered an error in the ontology. Ontology inspection tools, such as the RIO Protégé 4 plugin [MIS12], can assist ontology developers in detecting regularities and deviations in OWL ontologies.

2.2.3 Debugging Ontologies

In the context of OWL ontology engineering, the debugging stage involves the process of *detecting* an error in an ontology (e.g. an unsatisfiable class, or a wrong entailment), finding the *source* of the error, i.e. the information stated in the ontology which causes the error, and finally, *repairing* the ontology by modifying or removing (some of) the problematic information. Finding a cardinality-minimal repair may be considered the key principle of repair: Given an ontology, we want to find a *minimal* subset $\mathcal{R}$of $\mathcal{O}$, such that the entailment does not hold in $\mathcal{O} \setminus \mathcal{R}$, and such that there is no (strict) subset of R which fulfils this requirement. Minimal repairs are thought to ensure that no more information than necessary is lost from the ontology while also requiring the least effort from a user; as we will see, however, this is often not the case.

The repair process can be performed manually, using a reasoner as a tool to classify the ontology, then searching the class hierarchy for unwanted entailments (potentially aided by a suitable visualisation in an ontology editor, such as Protégé 4 which arranges all unsatisfiable classes under the class Nothing), tracing the source of the error by inspecting related statements in the ontology, and then modifying those statements that are considered to be incorrect.

Due to the potential size and complexity of OWL ontologies, this manual approach quickly leads to a tedious process of searching through the entire ontology. Moreover, it may lead to non-minimal repairs, where more information than required is removed or modified for the purpose of fixing the entailment. Anecdotes of ontology developers 'ripping out' parts of the ontology in order to remove an entailment clearly show that this approach is far from ideal and indicates a clear need for additional debugging support.

One of the first approaches to explaining subsumption in description logic knowledge bases was introduced by McGuinness [McG96] and Borgida [MB95], using proof-style explanations for entailments of the CLASSIC knowledge representation system [PSMB+91]. This proof-based approach was later extended by Borgida [BFH99] to KBS in the description logic $\mathcal{ALC}$. It was not until the introduction of *justifications*, however, that explanation support for description logic ontologies became a key aspect of ontology engineering research, spawning a substantial body of work and several ontology debugging tools.

2.3 Justifications for Entailments of Ontologies

Justifications are the most prominent form of debugging support for entailments of OWL ontologies. They originate from a seminal work by Schlobach and Cornet [SC03], who developed a strategy for *pinpointing* the causes of unsatisfiable classes in the medical ontology *DICE*. A justification $\mathcal{J}$ for an entailment η of an ontology $\mathcal{O}$ is a minimal subset of $\mathcal{O}$ which is sufficient for η to hold:

Definition 3 (Justification)**.** *$\mathcal{J}$ is a justification for $\mathcal{O} \models \eta$ if $\mathcal{J} \subseteq \mathcal{O}, \mathcal{J} \models \eta$ and, for all $\mathcal{J}' \subset \mathcal{J}$, it holds that $\mathcal{J}' \not\models \eta$.*

While Schlobach and Cornet focused on *MUPS*, Minimal Unsatisfiability Preserving Sub-TBoxes, in unfoldable $\mathcal{ALC}$ t-boxes, the concept of explanations based on minimal entailing subsets is applicable to arbitrary entailments. Kalyanpur et al [KPSH05] firstly extended the idea of *MUPS* to unsatisfiable classes in general OWL ontologies, presenting users with the *sets of support* (which were later named *justifications*) for an entailment.

The minimality of $\mathcal{J}$ implies that removing any one of its axioms *breaks* the entailment η. For any entailment η of $\mathcal{O}$, η itself is a justification, and there can be multiple justifications (exponentially many) for a single entailment.

Recall Example 3 which entails the class SickCat to be unsatisfiable. A justification for this entailment is the axiom set { axioms here }.In addition to limiting the number of axioms a user has to analyse when presented with a justification, OWL ontology editors also *order* and *indent* the axioms in justifications, which significantly increases

Arrange justification axioms properly

Figure 2.1: A screenshot of the *Explanation* tab in Protégé 4.

readability. Figure 2.1 shows the above justification as displayed in Protégé 4 with ordered and indented axioms.

The concept of justifications for description logic ontologies has been widely studied over the past decade [LN04, KPSH05, PSK05, Sch05a, Kal06, DS08], with particular focus on developing efficient algorithms for the computations of all justifications for a given entailment.

Bit more of an overview of justification research here.

Rather than generating the set of all justifications for an entailment, Baader and Penaloza propose the computation of a *pinpointing formula* [BPS07, BP08b, PS10] for explaining entailments in description logic ontologies. A pinpointing formula is a monotone boolean formula where each propositional variable represents an axiom in the ontology. The set of justifications for an entailment can then be derived from the pinpointing formula, corresponding to all valuations of the formula.

2.3.1 Computing Justifications

There exist various approaches for finding *one* or *all* justifications for a given entailment, with *find all* algorithms generally using a *find one* algorithm as a subroutine. We categorise these algorithms into *glass-box* and *black-box* approaches [KPSH05]: Glass-box techniques rely entirely on information provided by a DL reasoner, which requires modification of the reasoner internals in order to utilise it for justification computation. By contrast, black-box techniques only use the reasoner as an *oracle* to perform

entailment checks.

Glass-Box Techniques

Glass-box justification finding techniques rely on the modification of a tableaux reasoner to keep track of the axioms required for an entailment to hold, such as those axioms causing a *clash* in the tableaux procedure. More precisely, as the problem of explaining arbitrary entailments, e.g. subsumption between classes, can be reduced to a consistency check, this technique can be applied to find justifications for both unsatisfiable classes as well as arbitrary entailments. The tracking strategy applied in glass-box approaches is known as *tracing* [KPSH05], which is based on an algorithm first proposed by Baader and Hollunder [BH95]. A fair number of approaches to explaining entailments of description logic KBs uses glass-box techniques to present users with reasoner traces in the form of axiom sets, natural language, or some form of visualisation [SC03, LH05, Kwo05, MLBP06].

As some description logics require the use of 'sophisticated blocking conditions' 'baader08eb' in a tableaux algorithm, extending the respective tableaux algorithm with tracing to be used for pinpointing is not feasible. Thus, instead of using a standard tableaux tracing strategy, Baader and Penaloza [BP08a, BP10] propose a pinpointing technique using *looping tree-automata*.

Generally, glass-box techniques for finding single justifications are considered to be more efficient than black-box techniques, as the justifications are generated almost 'automatically' as a by-product of the classification process. And indeed, in an extensive analysis of various justification computation algorithms, Horridge [Hor11a] found that using a glass-box *find one* algorithm as subroutine can improve performance by an order of magnitude for some ontologies. However, given the heavy modifications required to integrate tracing with a reasoner, and the fact that currently only Pellet supports tracing, we may consider black-box techniques as a more accessible alternative to glass-box approaches.

Black-Box Techniques

Expand-Contract Approach A simple technique for finding one justification for an entailment η of an ontology $\mathcal{O}$ is the expand-contract approach [KPHS07]: In the *expansion phase* axioms from the given ontology $\mathcal{O}$ are incrementally added to an empty ontology $\mathcal{O}$', performing an entailment check after each addition until it is found that $\mathcal{O}$' entails η. In order to generate a minimal justification from $\mathcal{O}$', the *contraction phase* then prunes superfluous axioms from $\mathcal{O}$', performing an entailment check after each removal. This approach is known as *black-box* technique, as it only requires an out-of-the-box reasoner to perform the entailment checks. Optimisations of this algorithm

focus on reducing the number of expensive entailment checks by employing a *divide-and-conquer* [FS05, SFJ08] or a *sliding-window* technique [Kal06] in the contraction phase.

Hitting Set Tree Algorithm Schlobach [Sch05b] first proposed the use of Reiter's Hitting Set Tree algorithm [Rei87, GSW89] for the computation of MUPS and MIPS, i.e. justifications for unsatisfiable classes. The algorithm originates from the field of *model based diagnosis*, which describes the process of finding diagnoses for faults in a system comprised of components. A minimal conflict set is a minimal set of such components which causes a system fault, whereby a diagnosis is a hitting set across the set of minimal conflict sets; minimal conflict sets and diagnoses correspond to our notion of justifications and minimal repairs, respectively.

Reiter's algorithm constructs a hitting set tree (HST) in order to find all minimal hitting sets over a given set of minimal conflict set in a diagnosis problem. The nodes in the HST are labelled with minimal conflict sets, and the edges are labelled with components (axioms). For the purpose of finding justifications, the HST algorithm is initialised by computing a single justification using any glass-box or black-box *find one* algorithm, which suffices to generate the tree for *all* justifications, as shown in [KPHS07].

Optimisations, such as *early path termination* and *justification reuse* [KPHS07] help ensure that the algorithm terminates in practical time. Horridge [Hor11a] showed that it is possible to compute all justifications for direct atomic subsumptions from over 90% in the ontologies in a diverse test corpus; for the remaining ontologies, however, the algorithm did not terminate in the given time due to the large size of the constructed HST, or due to a timeout on entailment checks.

Modularisation

A module $\mathcal{M}$ of an ontology $\mathcal{O}$ is a subset of $\mathcal{O}$ which contains all axioms that are relevant for some seed signature $\Sigma \subset sig(()O)$. There exist various types of modules and algorithms for efficient computation of a module for a given seed signature, see, for example [CPSK06, CHKS07, SSZ09]. It has been shown that *syntactic locality based modules* are *depleting*, which means that they contain all justifications for the signature of a given entailment [CHKS07]. Further, for any given seed signature, there exists a unique and minimal locality based module [CHKS08].

When used in the justification computation process, a syntactic locality based module $\mathcal{M}$ is computed for the signature of an entailment, and the justification computation deals only with $\mathcal{M}$ rather than the full ontology $\mathcal{O}$ [Sun08, DQJ09]. Typically, the number of axioms in a module is small compared to the whole ontology [Sun08], which leads

to reduced computational load both in the expansion and contract phases, as well as for entailment checks. A study by Suntisrivaporn et al [SQJH08] using randomly selected entailments from three fairly large and complex OWL ontologies (Galen, NCI, and Gene Ontology) found that the average size of syntactic locality based modules was only between 0.05% and 1.6% of the ontology. Consequently, the justification computation performance was drastically improved by a factor of around 1000, making it possible to compute justifications even for entailments on which the algorithm using the full ontology timed out. As the overhead for module computation is neglectable compared to the overall justification computation time, and due to the significant improvements obtained through modularisation, it is now considered a standard optimisation used in justification computation approaches for OWL ontologies.

2.3.2 Justification-Based Repair

While justifications offer ontology users a focused view on the subset(s) of the ontology which are relevant to an entailment, they do not provide much guidance as to which steps a user has to take in order to repair the erroneous entailment. Rather, the user is expected to analyse the axioms in the justification and identify a set of axioms $\mathcal{R}$ which constitutes a repair for the entailments. A repair over the set of justifications for an entailment η is defined as follows [Hor11a]:

Definition 4 (Repair)**.** *Given $\mathcal{O} \models \eta$, the set of axioms $\mathcal{R}$is a repair for η in $\mathcal{O}$ if $\mathcal{R} \subset \mathcal{O}$, $\mathcal{O} \setminus \mathcal{R} \not\models \eta$, and there is no $\mathcal{R}' \subset \mathcal{R}$ such that $\mathcal{O} \setminus \mathcal{R}' \not\models \eta$.*

More informally, a repair is a *hitting set* $\mathcal{R}$such that for each justification $\mathcal{J}$ for the entailment there is at least one axiom in $\mathcal{J}$ which is contained in $\mathcal{R}$. As there are multiple possible repairs for a given set of justifications, we may want to find a *cardinality-minimal* repair, that is, a repair $\mathcal{R}$ such that there is no repair $\mathcal{R}$' with fewer axioms than $\mathcal{R}$, as this implies removing or modifying the smallest possible number of axioms.

The quality of a repair does not only depend on the number of axioms to be removed or modified, but also on the amount of useful information in the ontology which is *lost* through such modifications. Thus, a suitable repair for an entailment may not be cardinality-minimal, but rather dependent on the *power* and *impact* [KPSC06] of the axioms in the justifications.

The *impact* [KPSC06] of an axiom in a justification is the number of entailments (from some clearly defined finite entailment set, such as the set of entailed atomic subsumptions) that are lost when removing the axiom from the ontology. Given a set of justifications for an entailment (or set of entailments), the *arity* [SC03] of an axiom is the number of justifications it occurs in, that is, the number of justifications that can be *broken* by removing the entailment.

Kalyanpur et al [KPSC06] approached the problem of finding a suitable repair by introducing a *ranking* on the axioms in justifications. Axioms are ranked according to their arity, their impact, manually specified test cases by a user, which is an extension to the default impact in order to ensure that specific entailments are preserved, provenance information about the axiom (author, source reliability, time added or modified) and usage of the terms in the axiom signature across the ontology. The repair tool in the Swoop [KPS+06] editors allows users to specify the weights of the various ranking features, then recommends the preservation or removal of high- or low-ranked axioms, respectively. While this kind of elaborate debugging support sounds promising, there have not been any in-depth user studies to confirm whether and to which extent ontology developers use the tool and how they benefit from it.

2.3.3 Understanding Individual Justifications

While the main focus of justification research has been on the performance of justification finding algorithms, in recent years, the issue of *understanding* justifications has also received some attention. We know that justifications can significantly reduce user effort by allowing them to focus on a small, *relevant* subset of an ontology when attempting to understand the causes of an entailment; and yet, justifications suffer from various problems when it comes to users understanding the justification and finding a suitable repair, as is outlined in the following sections.

Fine-Grained Justifications

By definition, a justification is a minimal entailing subset of an ontology; that is, a justification contains axioms as they are *asserted* in the ontology. This means that the axioms in a justification can contain *superfluous* parts, i.e. subexpressions which do not contribute to the entailment. While such superfluous parts can distract users from the actual cause of an entailment, making the justifications more difficult to understand, removing axioms with superfluous parts in the repair process also means a loss of information which might be valuable to the ontology.

In order to cope with the issue of superfluity, the notion of *fine-grained* justification was first introduced by Kalyanpur et al [KPC06]. This approach was based on the idea of re-writing justification axioms in a normalized form, then splitting the axioms across intersections and only preserving those axioms which are required for the entailment in question to hold. The strategy was implemented in the Swoop editor, using strike-out and colour highlighting to indicate superfluous expressions in axioms—a simple, yet visually effective technique. Lam et al [LPSV06] directly integrated the idea of fine-grained justifications into a repair tool, computing the impact of each axiom rewriting based on the modification of its *subexpressions*.

Based on these first attempts, Horridge et al [HPS08] firstly proposed a formal definition for *laconic* justifications: A laconic justification is a justification which does not contain any superfluous parts, with every subexpression being as *weak* as possible [HPS08]. This implies that every subexpression in a laconic justification is relevant to the entailment. The authors [HPS08] also describe a method to compute the *preferred* laconic versions of a justification which results in a unique correspondence between a justification and its laconic variants. In short, the process

- removes any subexpressions from axioms which are not relevant for the entailment to hold.
- derives a single subsumption axiom from an equivalence where possible.
- substitutes concept names with the top concept where possible.
- weakens numerical restrictions to the smallest number possible.

The following example illustrates several aspects of non-laconicity in a justification:

Example 4.

$$\mathcal{J}_1 = \{A \sqsubseteq B \sqcap \leq 2r.C, \exists r.C \sqsubseteq D\} \models A \sqsubseteq D$$

First, we can reduce the intersection in the first axiom to the single expression $\leq 2r.C$ without affecting the entailment. Second, this expression can then be weakened to $\leq 1r.C$. Third, the concept name C in both axioms can be substituted with the top concept. This results in the laconic justification $\{A \sqsubseteq \leq 1r.\top, \exists r.\top \sqsubseteq D\}$.

A survey [Hor11a] of 72 of OWL ontologies from the NCBO BioPortal[11] revealed that non-laconic justifications are prevalent across OWL ontologies; indeed, the majority of the surveyed ontologies (69 out of 72) contained at least some non-laconic justification for an atomic subsumption, with 29 ontologies containing over 50% non-laconic justifications.

Masking

Justification *masking* [HPS08, HPS10a] occurs when the actual number of *reasons* why an entailment holds differs from the number of justifications that can be found for this entailments. There exist different types of masking, which are mainly caused by axioms containing superfluous expressions, i.e. being non-laconic. The examples given in [HPS10a] focus on unsatisfiable classes; it can easily be shown, however, that masking also occurs for arbitrary entailments.

Internal masking describes a situation where there justification contains more than one reason why an entailment holds, which means that there exist multiple laconic versions of the justification.

[11] `http://bioportal.bioontology.org/`

Example 5.

$$\mathcal{J} = \{A \sqsubseteq B \sqcap C, B \sqcup C \sqsubseteq D\}$$

Example 5 shows a justification $\mathcal{J}$ for $A \sqsubseteq D$ which contains two reasons for the entailment. The laconic versions of $\mathcal{J}$ are:

$$\mathcal{J}_1 = \{A \sqsubseteq B, B \sqsubseteq D\}$$
$$\mathcal{J}_2 = \{A \sqsubseteq C, C \sqsubseteq D\}$$

External masking involves other axioms from the ontology which do *not* occur in any justifications for an entailment.

Example 6.

$$\mathcal{O} = \{A \sqsubseteq C \sqcap D, C \sqsubseteq D\} \models A \sqsubseteq D$$

In the ontology shown in Example 6 there exists only one justification for the entailment $A \sqsubseteq D$, namely the first axiom $A \sqsubseteq B \sqcap C$. It is clear to see, however, that the two axioms combined result in another reason for the entailment—but the justification comprising both axioms would violate the minimality condition for justifications.

Cross masking is similar to external masking with the difference that the axioms involved are from other *justifications* for the same entailment rather than non-justification axioms.

Example 7.

$$\mathcal{J}_1 = \{A \sqsubseteq C \sqcap D\}$$
$$\mathcal{J}_2 = \{A \sqcup C \sqsubseteq D\}$$

In addition to the two justifications $\mathcal{J}_1$ and $\mathcal{J}_2$ for the entailment $A \sqsubseteq D$, there exists a third reason why the entailment holds, which comprises parts of both justifications: $\{A \sqsubseteq C, C \sqsubseteq D\} \models A \sqsubseteq D$. Again, this third reason comprising both axioms cannot be a justification, as it is non-minimal.

Shared cores masking occurs when two *different* justifications have the same reason why an entailment holds, i.e. they only differ in their superfluous parts. This implies that the actual number of explanations for an entailment is *lower* than it appears.

Example 8.

$$\mathcal{J}_1 = \{A \sqsubseteq D\}$$
$$\mathcal{J}_2 = \{A \sqsubseteq D \sqcap \exists r.C\}$$

An ontology which contains the two above axioms has two distinct justifications $\mathcal{J}_1$ and $\mathcal{J}_2$ for the entailment $A \sqsubseteq B$. However, the actual reasons why the entailment holds are identical, as the laconic version of J_2 is identical to J_1.

Masking may cause problems for users attempting to understand or repair an entailment. When repairing a justification where internal, external, or cross masking occurs, a user might only notice and modify one of the reasons, expecting the entailment to no longer hold after the modification. That modification, however, will create *another* justification for the entailment—an effect which may be rather surprising to the user. Worse even, the justification may interact with several other axioms to create multiple justifications, a situation which has been found to occur in the NCI thesaurus [Hor11a]. Furthermore, when analysing justifications for the purpose of gathering ontology metrics, justification masking may lead to over- or under-counting the number of *actual* justifications in an ontology.

It is clear to see that ontology debugging tools need to support users in cases where masking through non-laconicity occurs. Dealing with superfluity in an explanation tool, however, makes it necessary to balance two opposing requirements: First, we want the user to be presented with the most concise explanation as to why an entailment holds, and avoid cluttering and distraction caused by superfluous parts. Second, in order to facilitate understanding and repair, the explanations should directly relate to the asserted axioms in the ontology. To date, there have been no successful approaches to OWL ontology debugging that met both requirements.

Justification-Based Proofs

In addition to superfluity, there exist other reasons why a justification can be difficult to understand. Justifications may contain structural patterns and constructors whose implications are unfamiliar or non-obvious to the user. A popular example from the *Movie* ontology illustrates how even a very small justification can be hard to impossible to understand for OWL experts [HP09, HPS10b]:

Example 9.

$$\begin{aligned}\mathcal{J} = \{&\mathsf{Person} \sqsubseteq \neg\mathsf{Movie}\\ &\mathsf{RRated} \sqsubseteq \mathsf{CatMovie}\\ &\mathsf{CatMovie} \sqsubseteq \mathsf{Movie}\\ &\mathsf{RRated} \equiv \exists\mathsf{hasScript}.\mathsf{ThrillerScript} \sqcup \forall\mathsf{hasViolenceLevel}.\mathsf{High}\\ &\mathsf{domain}(\mathsf{hasViolenceLevel}, \mathsf{Movie})\} \models \mathsf{Person} \sqsubseteq \bot\end{aligned}$$

Example 9 shows a justification for the unsatisfiability of the class Person in the Movie ontology. Axioms 2 to 5 in the justification entail that $\mathsf{Movie} \equiv \top$, which implies that $\mathsf{Person} \sqsubseteq \mathsf{Movie}$, which, in turn, contradicts the first axiom that states the disjointness of the classes Person and Movie. When presented with this justification, subjects tend to give up or question the correctness of the justification [HPS10b].

Justification-based proofs seek to address the problem of understanding such justifications by providing users with a more detailed explanation which includes not only the justification axioms, but also intermediate entailments such as $\mathsf{Movie} \equiv \top$ in the previous example. Such intermediate entailments are known as *lemmas*: A lemma λ is an intermediate entailment of a subset S of a justification $(\mathcal{J}, \eta)$ such that $(\mathcal{J} \setminus S) \cup \lambda$ is a justification for η over the deductive closure of $\mathcal{J}$. In this thesis, we will use a simplified variant of the definition of justification lemmas given in [HPS09]:

Definition 5 (Lemma)**.** *Let $\mathcal{J}$ be a justification for an entailment η. A* lemma *of $(\mathcal{J}, \eta)$ is an axiom λ for which there exists a subset $S \subseteq \mathcal{J}$ such that $\mathcal{J} \setminus S \cup \{\lambda\} \models \eta$ for $S \models \lambda$, and S is a minimal entailing subset over the deductive closure of $\mathcal{J}$.*

In a similar approach to justification-based proofs, Nguyen et al [NPPW12b] generate natural language proofs based on justifications by identifying frequent sub-patterns, or *rules*, in justifications which lead to such intermediate entailments, then translating the proof trees into natural language. They extracted patterns, restricted to laconic axioms and a maximum size of 4 axioms, from 500 OWL ontologies and identified the 57 most frequent patterns, which are then used as the basis for natural language proofs.

Cognitive Complexity

To date, there have only been few attempts at investigating the understandability of justifications for OWL users. While there exists a number of user studies to evaluate OWL debugging tools [KPSH05, Lam07], they focus on measuring the time and success rates of the debugging tools and do not offer any further insights into how users interact with the explanations. The first major study on the cognitive complexity of

justifications was carried out by Horridge et al [HBPS11b] for the purpose of evaluating a complexity model for justifications which was constructed based on an initial exploratory study [HBPS11a]. The study involved 14 students from an OWL class who were presented with a set of axioms and an entailment and were asked to answer whether the entailment followed logically from the axiom set or not. It was found that the complexity predicted by the model coincided partially with the error rate in this task, with some anomalies in the error rates caused by a) superfluity in a justification which the model did not account for, and 2) by a 'flaw' in the experiment protocol which meant test subjects could answer the question correctly, but for the wrong reason.

Building on their work on justification-based proofs, Nguyen et al [NPPW12a] present an investigation of the difficulty users have with understanding various subpatterns that occur in OWL justifications. The authors conducted a study on a 'mechanical turk' like web platform where test subjects were presented with natural language reasoning problem which was directly based on one of the 51 (out of 57 they identified) justification sub-patterns. Each problem was answered by around 50 people, with correct answers ranging from 100% for the easiest rule to only 2% for the most difficult one. While natural language based explanation is potentially highly effective for OWL novices, the resulting proofs are often very large and might be difficult to navigate and understand themselves.

2.3.4 Understanding Multiple Justifications

In many ontologies used in practice, we find that there exist multiple justifications for a single entailment. While the possible number of justifications per entailment is exponential in the number of axioms in the ontology, the average number of justifications found in OWL ontologies is comparatively small. In the case where we encounter multiple justifications, however, we are often faced with several dozen to several hundred justifications. When attempting to repair an ontology, such numbers are clearly not suitable for manual repair by a user; and even with the common repair tool which displays all justifications as a list, there is hardly any chance of producing a minimal repair.

Root and Derived Justifications

Root and derived justifications provide an additional form of explanation support which aims to assist users in the simultaneous repair of multiple entailments. While the concept was originally introduced as root and derived *unsatisfiable classes* [KPSH05], it can be easily extended to arbitrary entailments, as shown in [MMV10]: Given a set of entailments $\varepsilon_{\mathcal{O}}$ and their justifications $Justs(\varepsilon_{\mathcal{O}})$, a *derived* justification $\mathcal{J}_D \models \eta_D$ is a justification which is a superset of some other justification $\mathcal{J}, \eta$. Repairing $\mathcal{J}$

first (e.g. by removing or modifying an axiom) will also repair $\mathcal{J}_D$. Note that there may be additional reasons for η_D to hold, which are not covered by this process. A *root* justification is a justification $\mathcal{J}_R$ in $Justs$ which is not a superset of any other justification in $Justs$.

Definition 6 (Root and derived justifications)**.** *Let $\varepsilon_\mathcal{O}$ be a set of entailments of interest, and $\mathcal{J}$ a justification for an entailment η in $\varepsilon_\mathcal{O}$. $\mathcal{J}$ is called a* root *justification if there exists no justification $\mathcal{J}'$ for an entailment in $\varepsilon_\mathcal{O}$ such that $\mathcal{J}' \subset \mathcal{J}$, else, $\mathcal{J}$ is called a* derived *justification.*

Kalyanpur et al [KPSH05] carried out the first (and one of few) user study for explanation support in OWL, evaluating the debugging facilities of the ontology editor *Swoop* [KPS+06]. The 12 study subjects were randomly divided into four groups and were given the task to repair unsatisfiable classes in three OWL ontologies using one of three types of debugging tools (clash information from the reasoner and sets of support, root and derived unsatisfiable classes, both), with the fourth group receiving no debugging support at all. Perhaps unsurprisingly, the study found that the group using both the clash information, sets of support (i.e. justifications) and root and derived information performed best (i.e. fastest) in this task.

2.4 Other Approaches to Explanation and Debugging

2.4.1 Proofs

Besides justifications, formal proofs are considered to be the most prevalent alternative form of explanation for logic-based knowledge bases. One of the first approaches to explaining entailments in the CLASSIC system using proof-like structures was presented by McGuinness and Borgida [MB95]. The system omits intermediate steps and provides further filtering strategies in order to generate short and simple explanations. Borgida et al [BFH99] first introduced a proof-based explanation system for knowledge bases in the Description Logic $\mathcal{ALC}$. The system generates sequent calculus [Fit83] style proofs using an extension of a tableaux reasoning algorithm, which are then enriched to create natural language explanations. Aiming to 'provide the simplest explanation possible', the authors also introduce a *relevance* function which simplifies the proofs by omitting 'irrelevant' parts. The consensus of these proof-based approaches seems to be that good proofs ought to be 'as simple and short as possible', yet, there have been no formal investigations into how ontology developers interact with such proof-based explanations.

While proof presentation in other logics has been well studied (e.g. [FH88]), surprisingly few authors are concerned with the cognitive aspects of proof understanding.

There exist formal definitions of *obvious proof steps* [Dav81, Rud87], however, these are merely formulated with the goal of creating *short* proofs and do not investigate other sources of complexity for human readers. By comparison, Lingenfelder [Lin89] also considers the skill level of the proof reader to be crucial in order to determine what can be considered 'trivial' in a proof, and emphasises the need for a *user model* in order to provide suitable proofs.

2.4.2 Ontology Revision

Ontology revision as described by Nikitina et al [NRG12] follows a semi-automated approach to ontology repair, with a focus on factually incorrect statements rather than logical errors. In the ontology revision process, a domain experts inspects the set of ontology axioms, then decides whether the axiom is correct (should be accepted) or incorrect (axiom is rejected). Each decision thereby has consequences for other axioms, as they can be either automatically accepted (if they follow logically from the accepted axioms) or rejected (if they violate the already accepted axioms).

The proposed system determines the impact a decision has on the remainder of the axioms (using a ranking function), and presents high impact items first in order to minimize the number of decisions a user has to make. Conceptually, this approach is straightforward and easily understandable for a user, as the cognitive effort is reduced to a simple yes/no decision, and the tool attempts to minimize the number of decisions that need to be made. In order to debug unwanted entailments, e.g. unsatisfiable classes, the set of unwanted consequences can be initialised with those erroneous axioms. The accept/decline decisions are then made in order to remove those axioms which lead to the unwanted entailments.

The approach proposed by Shchekotykhin [SFRF12] and Rodler [RSFF12] follows a similar strategy of repairing an ontology by specifying positive and negative entailments. It is directly related to justifications, but rather than computing the set of justifications for an entailments which is then repaired by repairing or modifying a minimal hitting set of those justifications, the diagnoses (i.e. minimal hitting sets) are computed directly. The authors argue that justification-based debugging is feasible for small numbers of conflicts in an ontology, whereas large numbers of conflicts and potentially diagnoses pose a computational challenge.

As in the ontology revision tool proposed by Nikitina et al [NRG12], the user is required to specify a background knowledge (those axioms which are guaranteed to be correct) as well as sets of positive (P) and negative (N) test cases, such that the resulting ontology $\mathcal{O}$ entails all axioms in P and does not entail the axioms in N. The user is then presented with a set of diagnoses which are suitable with respect to the given constraints, and decides which repair to apply.

Both approaches show some advantages over computing justifications in terms of computational performance. However, the semi-automated repair presented in [NRG12] means that the user has no control over which axioms to remove or modify in order to repair the unwanted entailments. Further, neither of the approaches supports *understanding* why those entailments hold, as the users are not presented with the actual *reasons* why the entailments hold, but only see a sequence of axioms or a set of diagnoses, respectively. While Shchekotykhin [SFRF12] aims to reduce the total number of steps taken in the debugging process, using the accept/reject revision technique proposed by Nikitina [NRG12] might require a user to go through a long and tedious revision process when repairing a large number of errors, for example after conversion from some other ontology format into OWL.

2.4.3 OntoClean

While not a debugging strategy as such, OntoClean [GW02, GW04] provides a framework for making modelling decisions in the ontology development process, with a strong focus on correct subumption relationships. This is motivated by a misuse of subsumptions to express other relations than is-a relationships, for example part/whole relationships. OntoClean helps users to validate taxonomies by identifying *metaproperties* of each class, such as *rigidity*, *identity*, *unity*, and *dependency*. The framework then specifies constraints on subsumptions between classes with different properties, such as if a class A is anti-rigid, i.e. an instance of A can *cease* to be a member of the class, then every subclass of A must also be anti-rigid. By applying OntoClean guidelines when building taxonomies, ontology developers can prevent subsumption relationships which may later lead to modelling inconsistencies.

2.4.4 Ontology Comprehension

Ontology users often require support in understanding an ontology, or parts of it, for example, when integrating an existing ontology into a project, which requires the ontology developer to familiarise themselves with the structure of the adopted ontology. While not specifically aimed at ontology comprehension, debugging tools can also help users to get a better grasp of the classes and relationships in an ontology, thus improving understanding of the ontology.

Ontology visualisation tools, such as the OWLViz plugin[12] in Protégé 4, the CropCircles tool [WP06] and the, admittedly rather exotic, music score notation of Barzdins and Barinskis [BB07], offer ontology users support when exploring an ontology in order to gain an overview of the relationships between its classes.

[12] `http://www.co-ode.org/downloads/owlviz/`

Visualisation techniques can also be used for the purpose of *model exploration* [BSP09], which can user support in understanding *non-entailment*. Users may want to understand, for instance, why a class is not entailed to be subclass of some other — a fact that cannot be explained by justifications, as it is not possible to point out a particular subset of the ontology that does *not* entail something [HBPS08]. The idea underlying model exploration is that seeing (subsets of) the models of an ontology helps users understand the relations between its classes, which, in turn, may support them in understanding why an entailment holds or does *not* hold in the ontology.

2.5 Summary and Conclusions

In this chapter, we have laid out the foundations for the research presented in this thesis. We introduced the basic concepts of description logic knowledge bases, such as their syntax, semantics, and standard reasoning services. The Web Ontology Language OWL is based on the highly expressive description logic $\mathcal{SROIQ}(D)$, which allows the use of a wide range of constructors for expressing relations between classes and individuals in a domain, as well as the attributes of such relations. We discussed the scope of logical and non-logical errors which can occur in OWL ontologies and introduced justifications as the currently dominant form of debugging support for entailments of OWL ontologies. This was followed by a closer look at the strategies for computing justifications, as well as some of the applications of justifications for repairing errors in OWL ontologies. Finally, we gave an overview over debugging strategies for description logic knowledge bases which are not directly related to justifications or only make indirect use of them, finding that the majority of approaches do not consider cognitive aspects of how users interact with explanation.

While multiple justifications for entailments of OWL ontologies have been acknowledged as a phenomenon to be found in OWL ontologies, the main focus of research thus far has been the efficiency of computing multiple justifications. In contrast, the issue of how users can find a suitable repair once those justifications have been computed has been largely neglected. There exist some approaches to making individual justifications easier to understand, such as laconic justifications and justification-based proofs, but the specific problem of debugging and repair in the presence of multiple justifications has not been addressed in prior research, except for some work looking at root and derived unsatisfiable classes. Root and derived unsatisfiable classes provide a first level of additional support for a particular relationship between multiple justifications for multiple entailments; yet, there have been no investigations into whether root/derived relationships are indeed a common feature of OWL justifications, what steps can be taken when this feature is not present, and how users can cope with multiple justifications for *single* entailments. Besides, while root and derived unsatisfiable classes

indicate that there exist structural relationships between justifications, there have not been any investigations into other such relationships and whether they can be exploited for debugging.

In summary, there does not seem to be a good understanding of how prevalent multiple justifications are in ontologies used in practice, neither are there sophisticated enough coping strategies to help ontology engineers deal with multiple justifications efficiently. These insights motivate the research presented in this thesis, as there is an obvious need for further investigation of the occurrence of multiple justifications, the relations between them, and techniques for reducing both technical (the number of steps taken to achieve a goal) as well as mental (the cognitive complexity of the information presented to a user) user effort in the presence of multiple justifications.

Chapter 3

Entailments

Entailment is regarded as the 'key inference' of the Semantic Web [SPC+07]. While the entailment relation $\models$ is well defined for OWL ontologies, misleading nomenclature in ontology tools and anecdotal evidence show that there exist common misconceptions about entailments: First, it is often assumed that the set of entailments of an ontology is finite, and that it is possible to extract the set of *all* entailments of an ontology. Second, it is assumed that only non-trivial information is contained in the set of entailments, and tautologies such as $A \sqsubseteq A$ are not 'real' entailments. Third, the term entailments is used interchangeably with *inferences*, and the information that is asserted in the ontology is often not considered to be an entailment itself.

As the notion of *entailment* is central to our investigation of justifications, we need to be clear about which set of axioms we refer to when we talk about entailments. In this chapter we outline the usage of entailments in various applications and discuss the problems arising from these selective views. The above examples illustrate the need for a definition of entailment sets which restricts the number of entailments to a finite, relevant, and manageable set of axioms, which, at the same time, can be extended or reduced depending on a user's application needs. We introduce a flexible definition of *finite entailment sets*, which takes into consideration direct and indirect subclass relationships, tautologies, dealing with unsatisfiable and universal classes, and ontology imports. While this chapter lays the foundations for the work with justifications in the remainder of this thesis, it may also be regarded as a self-standing discussion with relevance to applications such as the OWL API and ontology editing tools.

3.1 Properties of Finite Entailment Sets

Recall that the entailment relation $\models$ in $\mathcal{SROIQ}$ is defined based on the formal semantics given by an interpretation $\mathcal{I}$. An ontology $\mathcal{O} = (\mathcal{T}, \mathcal{A})$ entails that a (possibly complex) concept C is subsumed by a concept D, written as $\mathcal{O} \models C \sqsubseteq D$ if $C^{\mathcal{I}} \subset D^{\mathcal{I}}$

for every model $\mathcal{I}$ of $\mathcal{O}$. Similarly, $\mathcal{O}$ entails that C is equivalent to D if $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{O}$. A concept C is entailed to be *unsatisfiable*, i.e. $\mathcal{O} \models C \sqsubseteq \bot$ if $C = \emptyset$ for every model $\mathcal{I}$ of $\mathcal{O}$. The set of entailments of an ontology is therefore the set of all axioms α such that $\mathcal{O} \models \alpha$.

3.1.1 Relevance and Tautologies

Even for inexpressive DLs, the set of axioms which are logically entailed by an ontology can be infinite; it only requires boolean constructors to create an infinite number of expressions which are trivially true. Take, for example, an ontology $\mathcal{O}$ with $sig(\mathcal{O}) = \{A, B\}$ and $\mathcal{O} = A \sqsubseteq B$, whereby $\mathcal{O}$ is in a description logic which allows conjunction (e.g. the inexpressive DL $\mathcal{HL}$). This minimal example already leads to an infinite number of entailments: The tautologies $A \sqsubseteq \top$, $B \sqsubseteq \top$, $A \sqsubseteq A$, $A \sqsubseteq A \sqcap A$, ..., $B \sqsubseteq B$, $B \sqsubseteq B \sqcap B$, ..., $A \sqsubseteq B \sqcap B$, $A \sqsubseteq B \sqcap B \sqcap B$, ..., etc. are all entailments of $\mathcal{O}$.

While this example demonstrates a very obvious case of an infinite tautological entailment set which has absolutely no information value beyond the originally asserted axiom, even entailments which hold some information value may not be relevant to an ontology user. Given an ontology $\mathcal{O}' = \{A \sqsubseteq B, B \sqsubseteq \exists r.C\}$ in the description logic $\mathcal{EL}$, the set of entailments of $\mathcal{O}$ includes (amongst the entailments listed above) the axiom $A \sqsubseteq \exists r.C$. This axiom *does* indeed have some information value and, depending on the user's preferences, may be included in a view of an ontology's entailments. On the other hand, however, it is easy to see how including all non-tautological entailments of an ontology quickly leads to an entailment set which contains large amounts of irrelevant information.

In OWL tools such as the OWL API and Protégé 4, named classes that do not have a direct subsumer are generally included in entailment set as subclasses of $\top$. This ensures that all named classes in the ontology are present in the entailment set, however, entailments for classes which do have other named subsumers are quietly pruned, which may be confusing to a user. Further, in analytical applications, *counting* the number of entailments returned by these tools is simply misleading, as it adds tautological entailments for some classes while pruning others. Moreover, these entailments are of no use for analytical applications and simply skew the numbers of entailments.

3.1.2 Axiom and Expression Types

The axioms entailed by an ontology can be of various types and contain all expressions available in the chosen ontology language. As we have seen in the previous section, presenting a user with a finite, yet unrestricted set of entailment types is likely to yield a large and cluttered set, which is of no use to any human user dealing with an ontology.

When requesting a set of entailments from an ontology, we need to clearly specify the axiom type (subsumptions or equivalences) as well as the expression types we want to consider. The most commonly used type of entailment set is the set of atomic subsumptions (i.e. subsumption axioms that contain only atomic class names) which follow from an ontology, as these represent the ontology's class hierarchy, thus providing a natural starting point for exploring and understanding the ontology. Interestingly, while equivalences are equally relevant to the class hierarchy, they are often *not* included in the classification of an ontology. The OWL API provides an `InferredAxiomGenerator` interface whose implementations provide access to generators for all OWL axiom types, such as `InferredDisjointAxiomGenerator` and `InferredSubDataPropertyAxiomGenerator`, which return all inferred axioms of the specified types.

Beyond atomic entailments, we may also want to include entailments involving *complex* expressions, such as existential or universal quantifiers. Complex entailments are currently not supported in any of the previously discussed ontology development environments or the OWL API's `InferredAxiomGenerator`. From an engineering perspective, however, it can be useful to know, for example, whether a large number of named classes is subsumed by an expression such as $\exists r.B$ as this can justify the definition of a *named* class for the anonymous expression. As the number of potential expressions to check for is infinite, it seems reasonable to restrict the set of complex entailments to subsumptions and equivalences between an atomic concept and an expressions which does not contain any nested expressions, or, alternative, a maximal nesting depth of 1. Potential entailments include axioms such as $A \sqsubseteq \exists r.B$, $A \sqsubseteq \forall r.B$, $A \sqsubseteq B \sqcap C$, $A \sqsubseteq B \sqcup C$, $A \sqsubseteq \neg B$.

3.1.3 Asserted and Inferred Axioms

Every axiom that is asserted in an ontology $\mathcal{O}$ is naturally entailed by the ontology and therefore part of the set of entailments of the ontology. However, including asserted axioms in an entailment set may lead to large numbers of axioms which are considered to be *trivial* by a user as they are already explicitly 'written down' in the ontology. The main idea behind logic-based ontologies is the application of automated reasoning to make *implicit* knowledge visible; thus, the set of *inferred*, but not asserted, axioms may be of much higher relevance to a user. Most OWL tools (e.g. Protégé 4 and the OWL API) take this into account when computing the set of entailments of an ontology and return only entailments which are inferred, but not asserted.

While excluding asserted axioms from an entailment set is a reasonable choice in user-focused applications, this may lead to a loss of information in analytical applications, such as justification analysis. An entailment that is asserted in an ontology

may also hold for other, more complex reasons, i.e. it has justifications other than a self-justification. This may be purely accidental and a side-effect of other axioms in the ontology, or intentional as a result of ontology developers 'adding back' inferred entailments into the ontology. Making entailments explicit has two advantages: First, it may improve reasoner performance when classifying the ontology, as the reasoner will have to perform fewer subsumption checks; and second, the information will also be visible when there is no reasoner available, for example in a web-based ontology browser.

3.1.4 Transitivity

Transitivity causes some problems when attempting to define finite entailment sets. We generally treat the class hierarchy of an ontology as a directed graph, with nodes representing sets of named equivalent classes, and edges representing subsumption relationships. A *direct* subsumption between a class A and a class B is represented by an edge between the nodes labelled with A and B, respectively; this corresponds to a path of size 1. An *indirect* subsumption is a path between the two nodes of size greater than 1. The class graph can be either based on the *asserted* ontology, i.e. the class graph is constructed based on the axioms in the ontology, without the use of a reasoner, or the *inferred* ontology, which means that the class graph is built according to the subsumptions and equivalences returned by a reasoner. The set of all entailed atomic subsumptions, direct and indirect, then corresponds to the transitive *closure* of the inferred class graph, while the transitive *reduct* generates the set of entailed direct subsumptions.

There are several scenarios in which we may want to include or exclude indirect subsumptions in a finite entailment set. When presenting a set of entailed axioms to a user, we can assume that the user understands the principle of transitivity and only requires a small entailment set with *relevant* information; in such a case, it seems reasonable to exclude indirect subsumptions. On the other hand, if a user wants to be presented with *all* the information entailed by an ontology, it might be preferable to also include indirect subsumptions. For the purpose of computing and analysing justifications of an ontology, indirect subsumptions necessarily need to be included in the entailment set, as they may have relevant justifications which would otherwise be neglected.

With regards to *counting* the number of entailments of an ontology, for example to use the count as a metric for the *richness* of an ontology, care needs to be taken when dealing with the transitive closure of the inferred class graph. Example 10 shows how *removing* axioms from an ontology results in an *increase* in the number of entailments.


(a) Transitive reduct of the class graph of $\mathcal{O}$.


(b) Transitive reduct of the class graph of $\mathcal{O}$'.

Example 10.

$$\mathcal{O} = \{X_1 \sqsubseteq A, X_2 \sqsubseteq A, X_3 \sqsubseteq A$$
$$\mathsf{X_1} \sqsubseteq \mathsf{B}, X_2 \sqsubseteq B, X_3 \sqsubseteq B, A \sqsubseteq B\}$$

Given the above ontology, the number of entailed atomic subsumptions, based on the transitive reduct of the inferred class graph (shown in Figure 3.1a), is 4, as the indirect subsumptions between X_i and B are excluded from the set: $\varepsilon_{\mathcal{O}} = \{\ \mathsf{X_1} \sqsubseteq \mathsf{A}$, $\mathsf{X_2} \sqsubseteq \mathsf{A}$, $\mathsf{X_3} \sqsubseteq \mathsf{A}$, $\mathsf{A} \sqsubseteq \mathsf{B}\ \}$. Removing the axiom $A \sqsubseteq B$ from the ontology creates the class graph for the now modified ontology $\mathcal{O}$' shown in Figure 3.1b; as we can see, the transitive reduct of the graph has *more* edges than the graph for $\mathcal{O}$. Accordingly, the set of entailments based on the transitive reduct now contains 6 entailed axioms: $\varepsilon(\mathcal{O}') = \{\ \mathsf{X_1} \sqsubseteq \mathsf{A}$, $\mathsf{X_2} \sqsubseteq \mathsf{A}$, $\mathsf{X_3} \sqsubseteq \mathsf{A}$, $\mathsf{X_1} \sqsubseteq \mathsf{B}$, $\mathsf{X_2} \sqsubseteq \mathsf{B}$, $\mathsf{X_3} \sqsubseteq \mathsf{B}\ \}$. Thus, contrary to our intuition, the removal of an axiom has led to a *larger* entailment set.

Interestingly, the `InferredSubClassAxiomGenerator` implemented by the OWL API exhibits exactly this behaviour; and while this non-monotonicity of entailment counts is not *wrong*, it is certainly confusing and counter-intuitive to a user who has no way of specifying and understanding the exact settings of an entailment generator.

3.1.5 Equivalent Classes

Dealing with equivalent classes (or properties) as entailments is not as straightforward as it may seem, as both representing equivalent classes and subsumptions between equivalent classes and other classes as a set of axioms requires various choices.

Assume the classes A, B, and C are all entailed to be equivalent by some ontology $\mathcal{O}$. In the class graph of $\mathcal{O}$ this can be easily expressed by a node which is labelled with the three class names. In OWL applications, however, users are generally presented with a

list of entailed *axioms* rather than a graph visualisation. We now have to choose how to represent the equivalence between these three classes in a set of equivalence axioms: Do we choose 1) an n-ary EquivalentClasses axiom, which is possible in OWL, 2) an exhaustive set of binary equivalence axioms in order to correspond with DL notation (which only allows binary equivalence axioms), or 3) do we choose a set of *pairwise* axioms which suffices to represent the relation, such as $\{ A \equiv B, B \equiv C \}$?

Again, the decision how to represent equivalent classes depends on the application of the entailment set. In an OWL context, an n-ary EquivalentClasses axiom is certainly the most user-friendly way of representing a set of equivalent classes. When analysing justifications, the set of binary equivalent classes will capture all justifications, while a set of representative axioms by be suitable in a user-facing application which only supports binary equivalence axioms. As there are potential applications for all these alternatives, it is important to give the user (or application developer) the choice to select a suitable representation.

Further, representing subsumption relationships in a class graph between nodes labelled with multiple equivalent classes poses yet another challenge when attempting to generate a finite set of *axioms*. Given a set of equivalent classes A, B, and C which are all subsumed by a common superclass D, how can we represent this relation in a set of axioms?

Yet again, there are multiple approaches: 1) Every class in the set of equivalent classes creates a new subsumption axiom $A \sqsubseteq D$, $B \sqsubseteq D$, $C \sqsubseteq D$. If the chosen entailment set does *not* include equivalent classes, this approach may be suitable, as it explicitly lists the subsumption relations between the classes and its superclass.[1] The disadvantage of this strategy is a fast growth of subsumption axioms if both nodes contain multiple equivalent classes. 2) When including equivalent class axioms in the entailment set, it may suffice to select a *representative* from each of the nodes representing the sub- and superclasses, respectively. The knowledge of the equivalence relations and the subsumption between two of the classes will then be sufficient information for a user to infer that the other subsumptions follow. 3) Another, less straightforward approach, would be to attempt to represent all equivalent classes in a node by one newly generated class name, for example by concatenating the names of the subclasses and presenting the user an axiom of the type $A, B, C \sqsubseteq D$. This is obviously not standard OWL notation and might be confusing to a user, yet, it conveniently captures both the equivalence as well as the subsumption relations in a single axiom.

[1] Excluding the equivalent class axioms in this situation may lead a user to have an incorrect 'mental model' of the ontology graph in which the three subclasses are not equivalent.

3.1.6 Bi-Directivity

Another issue to consider when computing finite entailment sets is the notion of strict vs non-strict subsumptions. A strict subsumption is an axiom $A \sqsubseteq B$ such that $\mathcal{O} \models A \sqsubseteq B$ and $\mathcal{O} \not\models B \sqsubseteq A$, i.e. the classes are not equivalent, whereas a non-strict subsumption is an axiom $A' \sqsubseteq B'$ where $\mathcal{O} \models A' \equiv B'$.

By default, the OWL API excludes non-strict subsumptions from the set of entailed atomic subsumptions. This may cause confusion in cases where a user knows that class A should be subsumed by B, but is not aware of reasons for the equivalence of the two classes; again, the subsumption simply appears to be missing from the entailment set. Including non-strict subsumptions therefore seems reasonable when a user requests only subsumption axioms in an entailment set. On the other hand, when including equivalent classes in an entailment set, non-strict subsumptions should in turn be excluded from the set in order to avoid duplicating information in the resulting set.

3.1.7 Bottom and Top

An unsatisfiable class in an ontology is a class which is equivalent to $\bot$, i.e. it is mapped to the empty set in its interpretation. In ontology editors and in conversational use, however, unsatisfiable classes are frequently denoted as *subclass of bottom*. Similarly, a class which is entailed to be equivalent to $\top$ are denoted as superclass of $\top$ in ontology editors.

While this is obviously not wrong, it is also not completely accurate; in particular, when excluding non-strict subsumptions from an entailment set, including unsatisfiable classes as subsumptions is contradictory. In order to generate a sound entailment set, we need to be able to explicitly specify how to deal with classes which are equivalent to $\top$ or $\bot$. Displaying such entailments as subsumptions is certainly possible and has been the common usage, however, a user should at least be *aware* that this is only a one-sided view of the relationship.

3.1.8 Entailments in the Debugging Process

In the context of debugging an ontology we usually speak of modifying or removing axioms in order to remove *unwanted* entailments from the ontology. One of the main concerns here is to find a modification, i.e. repair, such that all unwanted entailments are removed, but at the same time as few *wanted* entailments as possible are lost from the ontology. In the most extreme case, a user could simply remove all axioms from an ontology, which would obviously have the desired effect of removing the unwanted entailments, but at the same time this would also remove any relevant information from the ontology. Finding a *minimal impact* modification is one of the key concepts

of belief revision; borrowing some notation from belief revision, we define the sets of wanted and unwanted entailments of an ontology:

Definition 7 (Wanted and Unwanted Entailment Sets)**.** *Given a finite entailment set* $\varepsilon_{\mathcal{O}} = \varepsilon^{+}_{\mathcal{O}} \cup \varepsilon^{-}_{\mathcal{O}}$

- $\varepsilon^{+}_{\mathcal{O}}$ *is the set of* wanted *entailments*
- $\varepsilon^{-}_{\mathcal{O}}$ *is the set of* unwanted *entailments.*

As the set of wanted (correct) entailments is likely to be significantly larger than the set of unwanted entailments, a reasonable approach in a debugging process would be to define first the unwanted entailments, as the wanted entailments then simply correspond to the remainder. The decision as to which entailments are unwanted lies with the user; a natural choice for $\varepsilon^{-}_{\mathcal{O}}$, for example, would be the set of unsatisfiable classes of an ontology. Given these two sets, we can make a clear distinction between modifications which lead to a positive effect (the removal of entailments in $\varepsilon^{-}_{\mathcal{O}}$) and those with a negative effect (the removal of entailments in $\varepsilon^{+}_{\mathcal{O}}$). Finding a balance between positive and negative removals is key in the ontology debugging process.

3.2 Dealing with Ontology Imports

Another issue that needs to be dealt with when extracting and counting entailments from an ontology is its import structure. An OWL ontology $\mathcal{O}$ that imports another OWL ontology $\mathcal{O}$' can have different kinds of entailments: those that hold in $\mathcal{O} \setminus \mathcal{O}$' (*native* entailments), those that are entirely from the imported ontology, i.e. they hold in $\mathcal{O}' \setminus \mathcal{O}$ (*imported* entailments), and those that hold in $\mathcal{O} \cup \mathcal{O}'$ but not in $\mathcal{O} \setminus \mathcal{O}'$ (*mixed* entailments).

When performing analytical tasks on a corpus of ontologies, disregarding the issue of imported entailments may in fact lead to significant distortions: Imagine a scenario where each ontology $\mathcal{O}_i$ in a test corpus imports another ontology $\mathcal{O}$', for instance an upper-level ontology. A finite entailment set of each of $\mathcal{O}_i$ would also include all entailments of $\mathcal{O}$', which would skew any analysis of the corpus towards the entailments of $\mathcal{O}$'. We encountered this situation when analysing a snapshot of the BioPortal corpus, in which a number of ontologies had exactly the same set entailments, as they were all importing the Basic Formal Ontology[2] (BFO).

In order to solve this issue, we propose a classification of entailment types based on the *origin* of an entailment which is determined by the set of its justifications. We use the following naming conventions:[3]

[2] `http://www.ifomis.org/bfo`

[3] `http://www.w3.org/TR/owl2-syntax/#Imports`

- $\mathcal{O}_{root}$ denotes the ontology *document* we are analysing, e.g. the .owl file that has been loaded into an ontology editor.
- $\mathcal{O}$ is the *import closure* of $\mathcal{O}_{root}$, i.e. the ontology resulting from transitive closure of the direct imports of $\mathcal{O}_{root}$ and of the ontologies it imports.
- $\mathcal{O}$' denotes an ontology in the import closure of $\mathcal{O}_{root}$ that is not the root ontology itself.

3.2.1 Type 1: Native Entailments

We may want to restrict the entailment extraction to the root ontology and discard all entailments that originate partly or entirely from the imported ontologies:

Definition 8 (Native Entailments)**.** *Let $\mathcal{O}$ the imports closure of an ontology $\mathcal{O}_{root}$. An entailment η is a* native *entailment of $\mathcal{O}$ if*

1. *$\mathcal{O} \models \eta$ and*
2. *for all α in all $\mathcal{J}$ where $\mathcal{J}$ is a justification for η in $\mathcal{O}$ it holds that $\alpha \in \mathcal{O}_{root}$.*

Note that while this restriction to native entailments may seem straightforward, it does not consider scenarios where an OWL ontology is simply split up over several files with different ontology IRIs: The imported files may be intended to be part of one and the same ontology, yet they are physically separate, which means their entailments are regarded as non-native according to our definition. The problem of determining fragments of an ontology could be solved either by using existing knowledge about the ontology imports structure, or by certain heuristics which group ontologies, for example, with similar filenames, such as *cheminf-core.owl*, *cheminf-algorithms.owl*, *cheminf.owl*, etc. in the case of the Chemical Information Ontology [KDLD08]. This raises further questions as to when ontology files can be considered fragments of a single ontology, and when they are complementary, but independently usable ontologies. For the time being, it is reasonable to say that the final decision whether entailments from imported ontologies (and from which imported ontologies) should be included or excluded lies with the user.

3.2.2 Type 2: Imported Entailments

While the entailments that originate purely from the imported ontology may not be relevant to an application, an analysis of the type and numbers of imported entailments provides information about the computational overhead they may cause when not excluding them from the entailment set:

Definition 9 (Imported Entailments)**.** *Let $\mathcal{O}$ the imports closure of an ontology $\mathcal{O}_{root}$. An entailment η is an* imported *entailment of $\mathcal{O}$ if*

1. $\mathcal{O} \models \eta$ *and*
2. *for all* α *in all* $\mathcal{J}$ *where* $\mathcal{J}$ *is a justification for* η *in* $\mathcal{O}$ *it holds that* $\alpha \notin \mathcal{O}_{root}$.

3.2.3 Type 3: Mixed Entailments

This type represents all entailments that are considered 'mixed' for at least one of the following reasons:

a) The entailment has at least one justification which contains axioms from both $\mathcal{O}_{root}$ and some $\mathcal{O}$'.
b) The entailment has some justification that comprises axioms from $\mathcal{O}_{root}$, and some justification that comprises axioms from some $\mathcal{O}$'.
c) The entailment has some justifications which are of type a), and some of type b).

3.2.4 Imported and Native Entailments in BioPortal

In a survey of a small sample of 42 OWL and OBO ontologies from the NCBO BioPortal [BHPS11], we found that imported entailments (restricted to entailed direct atomic subsumptions, excluding tautologies) did in fact have a visible effect on the number of entailments found in those ontologies. In this case the *Basic Formal Ontology (BFO)* [GSG04] contributed significantly to the skewing of entailment counts. BFO is an *upper-level ontology* which provides a conceptual framework for describing the spatio-temporal properties of entities.

7 ontologies in the sample corpus imported BFO, of which 3 had no native entailments at all, but only 70 imported entailments which all originated from BFO. A further 4 ontologies had 70 imported entailments from BFO, plus additional entailments which were either native or imported from ontologies other than BFO.

Additionally, a further 7 ontologies had imported entailments from ontologies other than BFO; for 5 of these, however, the imports could be attributed to the ontology being intentionally split up over several files with similar file names. Examples of these included the Chemical Information ontology mentioned above, which had 1 native entailment, 72 imported entailments from an ontology titled 'cheminf-external', and 3 entailments from 'cheminf-core', or the *Semanticscience Integrated Ontology* (SIO),[4] which had imported entailments from an ontology called *sio-core.owl*.

Finally, 28 ontologies did not have any imported entailments at all, which could be either due to them having no imports, the imported ontology having no entailments that matched our criteria, or missing imports, which were ignored in the pre-processing stage when downloading the ontologies from BioPortal.

The example of the BioPortal ontologies shows how an analysis or comparison of ontologies based on the number of entailments and justifications needs to pay attention

[4] `http://code.google.com/p/semanticscience/wiki/SIO`

to their import structure; in the case of the BFO imports, the 3 ontologies which had only entailments from BFO would have appeared to have exactly the same number of entailments and justifications, and thus might have been wrongfully classified as similar in terms of their expressivity and inferential power.

3.3 A Notation for Finite Entailment Sets

Table of names for 'constructors' for finite entailment sets.

3.4 Examples for Use of Entailments in OWL

A number of OWL applications provide methods for generating the 'inferred' version of an ontology, or offer views of 'selected entailments'. From working with these tools, we have found that the notion of *entailments* does not have a consistent interpretation across different applications. In this section, we briefly outline some of the examples for the use of entailments in OWL applications, which shows how they would benefit from a *flexible* definition of entailment sets.

3.4.1 Inferred Ontology Generation in the OWL API

The OWL API provides the convenience class `InferredOntologyGenerator`, which allows users to 'fill' a new ontology with the desired type of entailments, such as inferred atomic SubClass axioms and ClassAssertions. By default, this method only retrieves the *direct* named superclasses of a named class when using `InferredSubClassOfAxiomGenerator`. Additional properties, such as how to deal with non-strict subsumptions, and handling of imported ontologies, cannot be specific. While this method provides a common basis for computing the *inferred ontology*, it does not offer any flexibility for the user to specify which relationships should be included in the output.

3.4.2 Presenting Entailments to End-Users

The ontology editor Protégé 4 for instance comes with a 'selected entailments' tab which shows a list of atomic SubClassOf, SubPropertyOf and Type (class assertion) axioms. The tool also offers the option to 'Save inferred axioms as ontology' which saves asserted and inferred axioms as a new OWL ontology. Similarly, Top Braid Composer offers to 'Save [the] inference graph' as a new file. None of the editor offers any further explanation to how these entailments were extracted in the classification process. It may even seem surprising to the end-user that some trivial axioms, such as $A \sqsubseteq \top$, are displayed in Protégé 4's 'selected entailments' panel (shown in Figure 3.1) for *some* classes, while others are missing. A more transparent and modifiable view of

Figure 3.1: Screenshot of the 'Selected Entailments' tab in Protégé 4

such entailments could support users in exploring the class hierarchy when attempting to understand entailment relations in the ontology.

3.4.3 Ontology Publishing

Ontologies that are available on the web may be published as 'compiled' versions, which include the ontology and its entailments of some description. The OWL version of the National Cancer Institute (NCI) Thesaurus, for example, 'includes inferred relationships'.[5] There is, however, no definition of what is regarded as an inferred relationship, how these relationships are determined, and what the selection criteria is. This may leave users wondering what kinds of information they are dealing with, and what implications this has for their understanding of the ontology.

3.4.4 Metrics and Analytical Applications

Analytical applications that consider the number and type of entailments in order to infer ontology metrics benefit from clearly defined entailment set, as the basis of the measurements, i.e. *what exactly* is measured, is well defined and independent

[5] `http://evs.nci.nih.gov/ftp1/NCI_Thesaurus/ReadMe.txt`

from a particular implementation or *individual modifications* of results provided by the OWL API. From anecdotal evidence we know that developers of analytical tools frequently modify the results of the OWL API's `InferredAxiomGenerator` for use in their application, for example by removing entailments of the type $A \sqsubseteq \top$. By providing a definition of the type of the entailment set to be extracted based on the set of criteria specified above, we can ensure transparent and consistent measurements across different applications.

One possible application for entailment sets is the use of entailment counts as a metric for the *inferential power* of an ontology. As there is no formal definition of inferential power, the term is commonly used to refer to the expressiveness of an ontology which can be measured based on 'how many inferences' the statements in the ontology produce. Given two ontologies of equal size (equal numbers of axioms and signature size), the more 'powerful' ontology will have more inferred-only entailments, whereas the 'weaker' ontology may have, for example, only its asserted axioms as entailments.

3.4.5 Sample Entailment Sets

The following examples demonstrate the effect of different criteria on the size and types of axioms of finite entailment sets for a small ontology.

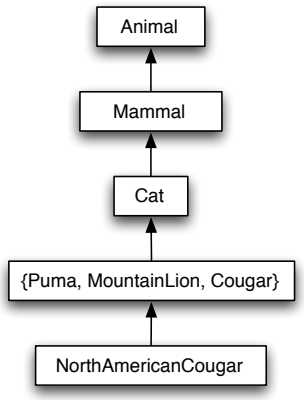
Example 11 (Toy ontology)**.**

$\mathsf{NorthAmericanCougar} \sqsubseteq \mathsf{Cougar}$	$\mathsf{Mammal} \sqsubseteq \mathsf{Animal}$
$\mathsf{Cougar} \equiv \mathsf{MountainLion}$	$\mathsf{Puma} \equiv \mathsf{Cougar}$
$\mathsf{Puma} \sqsubseteq \mathsf{Cat}$	$\mathsf{Cat} \sqsubseteq \mathsf{Mammal}$

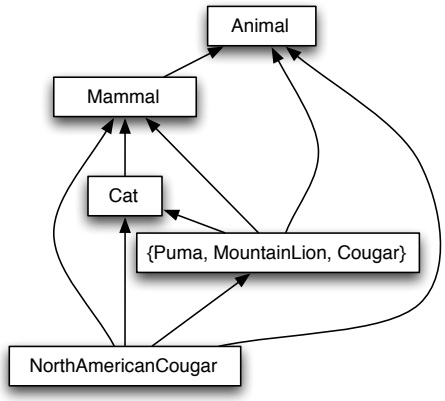
The inferred and asserted class graphs of this ontology are shown in Figures 3.2a and 3.2b, respectively.

Transitive Closure

The entailment set created from the transitive closure of the class graph, including asserted axioms and using an exhaustive representation of equivalent classes, make explicit the subsumption and equivalence relationships between every single class in the ontology. The set shown in Example **??** is the largest finite entailment set to be extracted from the class graph which does not contain any tautologies. The alternative variant of this set excluding asserted subsumption axioms simply discards the six axioms that occur in the original ontology, yielding a set of 12 axioms.



(a) Asserted class graph.



(b) Inferred class graph.

Example 12 (Transitive closure, including asserted, 18 axioms)**.**

Cougar $\equiv$ MountainLion	Cougar $\equiv$ Puma
MountainLion $\equiv$ Puma	Puma $\sqsubseteq$ Cat
Puma $\sqsubseteq$ Mammal	Puma $\sqsubseteq$ Animal
MountainLion $\sqsubseteq$ Cat	MountainLion $\sqsubseteq$ Mammal
MountainLion $\sqsubseteq$ Animal	Cougar $\sqsubseteq$ Cat
Cougar $\sqsubseteq$ Mammal	Cougar $\sqsubseteq$ Animal
NorthAmericanCougar $\sqsubseteq$ Puma	NorthAmericanCougar $\sqsubseteq$ MountainLion
NorthAmericanCougar $\sqsubseteq$ Cougar	NorthAmericanCougar $\sqsubseteq$ Cat
NorthAmericanCougar $\sqsubseteq$ Mammal	NorthAmericanCougar $\sqsubseteq$ Animal

Such a set of entailments is not suitable for user-facing applications, as it is very large and contains a lot of information which can be easily inferred by a human user who is familiar with transitivity. For the purpose of computing justifications, however, this set set seems most appropriate, as it guarantees that all subsumption relationships are captured.

Transitive Reduct

The entailment set based on the transitive reduction of the class graph uses representative elements from each node as well as a pairwise representation of equivalent classes to produce a *minimal* image of the class hierarchy.

Example 13 (Transitive reduct, including asserted, 6 axioms)**.**

NorthAmericanCougar $\sqsubseteq$ Cougar	Cougar $\equiv$ MountainLion
MountainLion $\equiv$ Puma	Puma $\sqsubseteq$ Cat
Cat $\sqsubseteq$ Mammal	Mammal $\sqsubseteq$ Animal

This entailment set contains all the information that would be necessary for a user to understand (or be able to infer) the relationships in the ontology. The class representing the node labelled with the equivalent classes {Cougar, Puma, MountainLion } was selected to be Puma, as the resulting axiom is part of the asserted ontology; therefore, in this example, the transitive reduct of the inferred class graph coincides with the asserted ontology. Choosing any of the two other classes would have been appropriate, too, but would have resulted in a non-minimal entailment set.

3.4.6 Example for Analytical Applications

Example: differences between entailment sets, justification generation...

NCI Thesaurus Diff

Justification Computation

Justification computation gets harder for indirect subsumptions.

3.5 Summary and Conclusions

In this chapter, we presented a discussion of the issues surrounding finite entailments sets for OWL ontologies. While the classification and computation of inferences is a standard reasoning task in the ontology engineering process, there exist ambiguities inaccuracies in ontology tools as to what constitutes the 'set of entailments' of an ontology. We highlighted various properties which are needed to clearly specify a non-ambiguous finite entailment set of an OWL ontology: How to deal with tautologies, which axiom and expression types to include, whether to include asserted axioms, how to deal with transitive and non-strict subsumptions, and how to make the transition from nodes and edges in an abstract class graph to concrete sets of *axioms* which are presented to a user or used in an OWL application. We also highlighted the problem of *imported entailments* and how these can skew ontology metrics based on entailments, which was demonstrated using examples from a corpus of bio-medical ontologies. The examples found in this chapter illustrate the wide scope of different entailment sets that

can be specified depending on the application, ranging from minimal representations which are suitable for user-facing applications, to exhaustive, yet finite, sets which may be used for analytical purposes.

The outcome of the work presented in this chapter is a deeper insight into what we mean by the 'set of entailments' of an ontology, alongside a set of properties which can be used to precisely define a finite entailment set that is fit for a specific purpose. The main benefit of this set of properties is that it is defined from an *application* perspective, which makes it easy to understand for a user what effect individual choices, such as whether to include indirect subsumptions, will have on the resulting entailment set. Further, the implementation of such an entailment selector is straightforward, with the only challenges being the function to select a representative class from a node of equivalent classes, while the OWL API already provides a method to transform an n-ary equivalence axiom into a set of binary (pairwise) equivalent classes axioms. The understanding and tools gained here lay the foundations for our discussion on justifications, and the relations between justifications for entailment sets, which will be presented in the following two chapters.

Chapter 4

The Justificatory Structure of OWL Ontologies

Thus far, we have only considered justifications as individual entities in relation to a single entailment. While this view reflects the way justifications are handled and represented in current ontology explanation tools, it entirely neglects the rich relations between *multiple* justifications for both single and multiple entailments of OWL ontologies. The majority of ontologies used in practice contain some entailment which has more than one justification, with many ontologies generating a few dozen and up to several hundred justifications for a single entailment.

Several properties of justifications, such as the number of justifications for an entailment, the size of justifications in an ontology, to which extent the ontology contains entailments which are entailed only be themselves, and which proportion of the ontology participates in justifications, all give us an insight into structural relationships in the ontology beyond standard metrics; we can say that the justifications make the *implicit* structure of an ontology *explicit*.

Further, justifications for a single or multiple entailments are not disjoint subsets of an ontology, but they often *overlap* to a certain extent, sharing one or more axioms between them. This is relevant for both understanding as well as repairing the entailments of such justifications: Shared axioms may[1] lead to a smaller hitting set, and therefore to a smaller repair for a set of entailments. Further, shared axioms may indicate common lemmas, i.e. intermediate entailments, which can assist users in understanding multiple justifications at the same time rather than treating each one independently, thus reducing the effort required to deal with multiple justifications.

In this chapter, we will introduce the notion of *justificatory structure* of an OWL ontology. We lay out the motivation for analysing the relations between justifications in

[1]It is important to point out that a shared axiom does *not* necessarily have to be an incorrect axiom; therefore, it is not guaranteed to be part of the hitting set that leads to a repair.

an ontology and define various structural aspects which may be of interest for ontology development and analysis. We introduce a graph-based representation of justifications and entailments in an ontology and conclude the chapter with a description of the generation and implementation of such a *j-graph*.

4.1 Representing Justifications as J-Graphs

A justification $(\mathcal{J}, \eta)$ is defined with respect to a single entailment η; in order to describe the set of all justifications for a single entailment or all entailments in an entailment set $\varepsilon_\mathcal{O}$, we introduce the notion of *justification sets*:

Definition 10 (Justification set)**.** *Given an ontology $\mathcal{O}$ and an entailment η (entailment set $\varepsilon_\mathcal{O}$), the justification set $Justs(\eta)$ ($Justs(\varepsilon_\mathcal{O})$) is the set of all justifications $\{\mathcal{J}_1 \dots \mathcal{J}_m\}$, $\mathcal{J}_i \subseteq \mathcal{O}$, for η (the axioms in $\varepsilon_\mathcal{O}$).*

Further, we define the set of all axioms occurring in all justifications for a particular entailment set:

Definition 11 (Justification axioms)**.**

$$JustAx(\varepsilon_\mathcal{O}) = \{\alpha \mid \textit{there is a } \mathcal{J} \in Justs(\varepsilon_\mathcal{O}) \textit{ s.t. } \alpha \in \mathcal{J}\}$$

Based on the above definitions of justification sets and justification axioms, we can now define the justification graph of an ontology $\mathcal{O}$ with respect to an entailment set $\varepsilon_\mathcal{O}$. A justification graph (*j-graph*) $G(\varepsilon_\mathcal{O})$ is a directed graph whose set of vertices is the union of the set of axioms $\varepsilon_\mathcal{O}$ which are entailed by $\mathcal{O}$ and the set $JustAx(\varepsilon_\mathcal{O})$ of axioms that participate in justifications for these entailments, together with the set of all justifications $Justs(\varepsilon_\mathcal{O})$. The graph does not contain axioms in $\mathcal{O}$ which are neither in the entailment set, nor in $JustAx(\varepsilon_\mathcal{O})$, as these would not have any incoming or outgoing edges, thus only adding clutter without any information content.

Definition 12 (Justification graph)**.**

$$\begin{aligned} G(\varepsilon_\mathcal{O}) &= (\varepsilon_\mathcal{O} \cup Justs(\varepsilon_\mathcal{O}) \cup JustAx(\varepsilon_\mathcal{O}), E_1 \cup E_2) \textit{ where} \\ E_1 &= \{(u, v) \in \varepsilon_\mathcal{O} \cup JustAx(\varepsilon_\mathcal{O}) \times Justs(\varepsilon_\mathcal{O}) \mid u \in v\}, \\ E_2 &= \{(v, w) \in Justs(\varepsilon_\mathcal{O}) \times \varepsilon_\mathcal{O} \mid v \in Justs(\mathcal{O}, w)\}. \end{aligned}$$

Figure 4.1 shows a j-graph of the small sample entailment and justification set in Example 14.

Example 14. *example from BP ontology*

Figure 4.1: An example of a j-graph for justifications and entailments.

Side Remarks

1. Recall that the number of justifications for an entailment can potentially be exponential in the number of axioms in $\mathcal{O}$, and that it might not be practical to compute *all* justifications for an entailment set, thus resulting in an *incomplete* j-graph. While this omission of information is obviously problematic for users attempting to find a repair for *all* justifications for a given entailment set, the only solution to computational problems is incremental repair. Therefore, we may consider the j-graph for an entailment set to be complete with respect to the justifications that can be computed in a given time frame, even if these are not all that exist in the ontology.

2. There exists a *unique* j-graph for every set of entailments, as the set of justifications for an entailment set is unique in an ontology (modulo incompleteness due to computational issues), and the edges in the j-graph follow from these unambiguous relations.

3. An axiom vertex in the graph with an in-degree ≥ 1 corresponds to an entailment in $\varepsilon_{\mathcal{O}}$, and an axiom vertex with an out-degree ≥ 1 corresponds to an axiom in $JustAx(\varepsilon_{\mathcal{O}})$.

4. While some j-graphs may be bipartite, an axiom vertex can be both in the set $JustAx(\varepsilon_{\mathcal{O}})$ and in $\varepsilon_{\mathcal{O}}$, i.e. have an outgoing edge to a justification vertex and an incoming edge from a justification vertex. Thus, a j-graph is not guaranteed to be bipartite.

This concludes the introduction of the j-graph structure for representing justifications. While a j-graph may not be necessary for an analysis of the justificatory structure of an ontology, having a graphical representation to describe structural aspects helps to provide a clearer picture of relations between axioms and justifications.

4.2 Justificatory Structure

Using the j-graph to describe relations between entailments, their justifications, and the axioms in the justifications, we can now discuss the various facets of the *justificatory structure* of an OWL ontology. In a first instance, the justificatory structure provides implicit ontology metrics; beyond simple *static* metrics, insights into the structure can support user understanding when coping with multiple justifications and/or multiple entailments. Further, in Section **??** we will introduce debugging techniques that are directly based on the interaction with the j-graph for a user-defined entailment set.

Note that in this section we refer to 'the entailments' and 'the justifications' of an ontology, which requires a more precise specification: Given an ontology $\mathcal{O}$, we fix a finite set of entailments $\varepsilon_{\mathcal{O}}$, then compute all justifications $Justs(\varepsilon_{\mathcal{O}})$ for these entailments. In what follows, we assume these entailments to be the set of native, direct and indirect (non-tautologic) atomic subsumptions which are entailed by $\mathcal{O}$, as this provides a clearly defined and natural starting point for exploring the properties of an ontology.

4.2.1 Properties of Justifications

Self-Justifications and Self-Supporting Entailments

Any justification which is simply the entailed axiom itself is classified as a *self-justification*:

Definition 13 (Self-justification)**.** *A justification $\mathcal{J}$ for an entailment η is a* self-justification *if $\mathcal{J} = \{\eta\}$.*

We distinguish between entailments which have a self-justification in addition to other, more complex, justifications, and entailments which have only a self-justification and no other justifications. These entailments are commonly referred to as *asserted*, but not *inferred*; however, in order to make avoid ambiguity caused by the word *inferred* (as an asserted axiom can also be inferred from an ontology), we denote such entailments as *trivial*, or, more descriptive,*self-supporting* entailments:

Definition 14 (Self-supporting entailment)**.** *An entailment η is* self-supporting *if $Justs(\eta) = \{\eta\}$.*

In a j-graph representation of the justifications and entailments, a self-justification corresponds to a cycle between an axiom vertex and a justification vertex. A self-supporting entailment is represented by such an axiom vertex which has no other incoming edges in addition to the cycle.

There are different reasons why an entailment might have a self-justification *in addition* to other justifications:

1. The entailment was not asserted in the ontology to begin with, but explicitly added after it was found to be inferred. This could be in order to improve reasoner performance, to make the entailment visible to the *developer* without using a reasoner (e.g. if the classification time is very high), or to make the inferences visible to *end-users* in an ontology browser interface which does not support reasoning.

2. The ontology modeller does not use a reasoner during the engineering process, thus is not aware that the subsumption is already entailed by the ontology, therefore explicitly asserting information which exists only implicitly.

3. The additional justifications are a side-effect of other axioms that were added to the ontology without the aim of causing the entailment.

Without additional information, such as axiom annotations or change logs of an ontology, it is not possible to tell the intentions of an ontology developer when adding an axiom which causes additional justifications or self-justifications. Moreover, developers may not even be *aware* of the full impact that a modification has on the ontology. We therefore treat self-justifications and additional justifications on a purely logical level, disregarding the reasons for those multiple justifications.

Justificatory Redundancy

Following on from the discussion of self-justifications and additional justifications, we can regard the number of justifications per entailment as an indicator of *justificatory redundancy* in the ontology; it demonstrates 'how often the same piece of information is expressed in different ways'. Justificatory redundancy is not to be confused with logical redundancy, as this would imply that it is possible to remove a set of axioms from the ontology without breaking any entailments.

In the j-graph, an axiom vertex with an in-degree > 1 has redundant justifications; thus the average in-degree of axiom-vertices is a metric for determining justificatory redundancy in an ontology.

Justfication Size

The *size* of a justification is the number of axioms it contains; this corresponds to the in-degree of a justification vertex in the j-graph. Justification size gives us information about the connectedness of an ontology: Large justifications are caused by signatures

which spread across multiple axioms, while small justifications indicate that the entities in the ontology are less connected.

The numbers and sizes of justifications are directly connected to the size of *depleting modules* in an ontology. Since a depleting module for the signature of an axiom contains all justifications for that axiom, the module grows with the number of justifications and the number of axioms in a justification for a given entailment. A comparison of the modular structure of an ontology and its justificatory structure follows in Section 7.2.3.

Activity

Given a finite entailment set of an ontology, we can define the *activity* of the ontology with respect to the entailment set. The activity corresponds to the total number of axioms that occur in justifications which are *not* self-justifications, that is, the size of the subset of the ontology which actively participates in inference:

Definition 15 (Activity)**.**

$$activity(\mathcal{O}, \varepsilon_{\mathcal{O}}) = \frac{\sum |\mathcal{J}_i|}{|\mathcal{O}|} \; where \; \mathcal{J}_i \in Justs(\varepsilon_{\mathcal{O}}), \; \mathcal{J}_i \; is \; not \; a \; self-justification.$$

I think I need to introduce some factor here to take into account the total size of the ontology, as large ontologies typically have a smaller proportion of axioms participating in justifications. I would think that the normal growth of axioms is a bit like a log graph - need to check the exact figures though.

4.2.2 Relations Between Justifications

Structural Regularities

Two types of patterns can be identified in the context of the justificatory structure, namely *structural isomorphism* between isomorphism and *graph surface patterns*.

Justification isomorphism describes a situation where the *axioms* within a set of justifications share an identical structure; for example, the two justifications $\{A \sqsubseteq \exists r.B, \exists r.B \sqsubseteq C \} \models A \sqsubseteq C$ and $\{X \sqsubseteq \exists s.Y, \exists s.Y \sqsubseteq Z \} \models X \sqsubseteq Z$ are isomorphic, as they have the same structure and only differ in the class and property names used. Justification isomorphism will be discussed in great detail in the next chapter; for now we will focus on structural properties of the j-graph and treat justification axioms as 'black boxes'.

A surface pattern is a structural similarity between sets of vertices and edges in the j-graph, such as subgraphs which match based on their vertex types and the numbers of ingoing and outgoing edges. Surface patterns in the j-graph reveal modelling similarities in the ontology, regardless of whether the axioms and expressions occurring in the pattern also interact in a similar way. Highlighting a pattern of this type may support user understanding of the modelling in the ontology, while it may also be an indicator for isomorphic justifications.

Components

A graph *component* is a subset of vertices and edges which forms a disconnected subgraph. The number of components of a j-graph provides a measure for the disjointness of justifications in the ontology. The disjointness of justifications strongly affects the justification computation process, which makes use of Reiter's Hitting Set Tree (HST) algorithm for diagnosis. Recall that in the HST, the vertices are labelled with justifications and the paths constitute hitting sets across the justifications in the tree, i.e. minimal repairs for the justifications. Optimisations for the HST algorithm are mainly based on closing a branch in the tree if the path to it is labelled with a superset of an existing path, which is not possible if the justifications are disjoint. This is thought to lead to a rapid growth of the HST and have significant negative effects on the performance of computing all justifications for an entailment. We will discuss the results of some experiments regarding this hypothesis in Chapter 7.

4.2.3 Properties of Axioms in Justifications

Axiom Arity and Impact

The arity and impact of a justification axiom refer directly to the effect the removal an axiom would have on a given ontology and entailment set.

The *arity*, also referred to as *power* or *frequency*, of an axiom is the number of justifications (for a given entailment set $\varepsilon_{\mathcal{O}}$) the axiom occurs in. In a j-graph, the arity of an axiom vertex corresponds to its out-degree; an axiom with an arity greater than 1 corresponds directly to a justification overlap of size 1 between the justifications the axiom occurs in. Removing an axiom with arity n from the ontology will break n *justifications* for the entailments in $\varepsilon_{\mathcal{O}}$. Given that there can exist multiple justifications for an entailment, the arity of the axioms in a set of justifications does not necessarily tell us how removal would affect the actual *entailments* of those justifications.

Add an example to make it clearer.

The *impact* of an axiom refers to the number of *entailments* in $\varepsilon_{\mathcal{O}}$ that are directly affected by the axiom. Removing an axiom with an impact of m from the ontology directly breaks m entailments in $\varepsilon_{\mathcal{O}}$. In a j-graph, the impact of an axiom vertex u is defined as the out-degree of every justification vertex v which has an incoming edge from u, i.e. where $(u, v) \in E_1$.

In the context of repairing a given set of entailments, we will need to consider the impact of axioms both with regard to the set of *unwanted* as well as the set of *wanted* entailments: Removing a high-impact axiom from an ontology may break a large number, or all, unwanted entailments (e.g. unsatisfiable classes), but it may also lead to the removal of relevant information. Referring to our definitions of wanted and unwanted entailment sets in Section 3.1.8, we can distinguish the *positive* and *negative*

impact of an axiom:

Definition 16 (Positive and Negative Impact)**.**

$$impact^{-}(\alpha) = |\{\eta_i\}| \ \ where\ \eta_i \in \varepsilon^{+}_{\mathcal{O}}, \alpha \in JustAx(\varepsilon^{+}_{\mathcal{O}})$$
$$impact^{+}(\alpha) = |\{\eta_i\}| \ \ where\ \eta_i \in \varepsilon^{-}_{\mathcal{O}}, \alpha \in JustAx(\varepsilon^{-}_{\mathcal{O}})$$

In words, the negative impact of an axiom α is the number of *wanted* entailments that are entailed by the justifications α occurs in, and the positive impact is the number of *unwanted* entailments that are entailed by the justifications α occurs in. Note that the plus and minus signs for positive/negative impact and wanted/unwanted entailment sets are swapped, i.e. $impact^{-}$ refers to $\varepsilon^{+}_{\mathcal{O}}$, and vice versa.

Arbitrary Justification Overlap

The concept of analysing overlaps and frequently occurring axiom groups in justifications for the purpose of finding a suitable repair was proposed by Schlobach in the earliest stages of justification-based explanation research [Sch05a]. Due to a lack of an efficient implementation, however, the author restricted the analysis of such *maximal arity cores* to single axioms.

The occurrence of such subsets gives rises to a hypothesis about their suitability for lemma generation: Justification subsets (i.e. justification overlaps containing multiple axioms) which occur in multiple justifications lead to a lemma, that is, an intermediate entailment that is not one of the axioms in the overlap, which is essential for the actual entailment to hold. Such a conclusion does not follow immediately for any given overlap, since it is quite possible for a set of axioms to occur in multiple justifications without there being an interaction between the axioms. Consider the two justifications in Example 15:

Example 15.

$$\mathcal{J}_1 = \{A \sqsubseteq B \sqcap C, B \sqsubseteq \exists r.D, \exists r.D \sqsubseteq E, E \sqsubseteq F\} \models dlax A \sqsubseteq F$$
$$\mathcal{J}_2 = \{A \sqsubseteq B \sqcap C, C \sqsubseteq \exists s.G, \exists s.G \sqsubseteq E, E \sqsubseteq F\} \models dlax A \sqsubseteq F$$

While $\mathcal{J}_1$ and $\mathcal{J}_2$ do overlap in two axioms $\{A \sqsubseteq B \sqcap C,\ E \sqsubseteq F\ \}$, there exists no lemma originating from *both* axioms which would be relevant to the entailment $A \sqsubseteq F$. In contrast, the shared axioms in Example 16 form a common lemma for both justifications:

Example 16.

$$\mathcal{J}_1 = \{A \sqsubseteq \exists r.B, \mathsf{domain}(\mathsf{r}, \mathsf{C}), A \sqsubseteq D, C \sqcap D \sqsubseteq E\} \models dlax A \sqsubseteq E$$
$$\mathcal{J}_2 = \{A \sqsubseteq \exists r.B, \mathsf{domain}(\mathsf{r}, \mathsf{C}), C \sqsubseteq F\} \models dlax A \sqsubseteq F$$

The axiom subset $\{ A \sqsubseteq \exists r.B, \mathsf{domain}(\mathsf{r}, \mathsf{C}) \}$ entails the lemma $A \sqsubseteq C$ which is relevant to the entailments of both justifications in the sense that if the lemma did not follow from these axioms, the entailments of $\mathcal{J}_1$ and $\mathcal{J}_2$ would not hold.

Arbitrary overlap is interesting in the context of justificatory structure as it indicates axiom sets which can be considered self-contained entities. Further, with respect to justification isomorphism, such overlaps lend themselves to lemma generation for lemma-isomorphism. With respect to the debugging of faulty entailments, lemmas of frequently occurring axiom sets may be presented to a user to support understanding by encouraging *chunking* of information, a concept which will be discussed in detail in Chapter 6. Note that our definition of lemmas only allows intermediate entailments which originate from a subset that can be substituted by the lemma. Thus, we can say that if there exists a lemma for an overlapping axiom set at all, the overlap can be considered relevant to the entailment. On the other hand, however, intermediate entailments which follow from subsets that cannot be substituted may be left out, even if they are relevant and potentially useful for understanding. Given that the substitution property of lemmas has clear benefits, this seems an acceptable tradeoff.

In the j-graph, arbitrary overlap between n justifications is represented by an intersection of the *incoming neighbours* of the justification vertices with a cardinality greater than 1. An intersection of size 1 corresponds to a single axiom with an arity of n, thus, we are interested in those overlaps which contain more than just one axiom. In order to efficiently determine arbitrary justification overlap, we make use of an algorithm from *Formal Concept Analysis*, which is described in Section 4.3.2.

Root and Derived Justifications

A special case of justification overlap are root and derived justifications, which we discussed in Section 2.3.4. Recall: A justification $\mathcal{J}$' is derived fr om a justification $\mathcal{J}$ if $\mathcal{J}$ (the root justification) is a strict subset of $\mathcal{J}$'. Due to the minimality of justifications, root and derived relations only occur between justifications for *multiple* entailments. In the j-graph, a root and derived relationship between two justification vertices is defined as a subset relations between the sets of incoming neighbours of the two vertices:

Definition 17 (Root and Derived Justifications)**.** *A justification vertex v is* derived

from a justification vertex v' if $N_{in}(v) \subset N_{in}(v')$, where $N_{in}(v) = \{u_i\}$ s.t. $(u_i, v) \in E_1$. A justification vertex which is not derived from any other justification vertex is a root *justification vertex.*

Note that root and derived justifications were initially introduced as root and derived unsatisfiable classes. However, as the concept holds for arbitrary entailments and justifications, we can speak of root and derived entailments in the same way.

Example 17.

$$\mathcal{O} = \{A \sqsubseteq \exists r.B, \exists r.B \sqsubseteq C, B \sqsubseteq D, \exists r.D \sqsubseteq E\}$$

In Example 17, the set comprising the first two axioms is a justification $\mathcal{J}_1$ for $A \sqsubseteq C$, while the set of all four axioms is a justification $\mathcal{J}_2$ for $A \sqsubseteq E$. Here, $\mathcal{J}_1$ is a root justification, and $\mathcal{J}_2$ is derived from $\mathcal{J}_1$. The two entailments are clearly related via the two axioms which constitute the root justification; we can say that $\mathcal{J}_1$ is the *root* of the entailments, while the remaining two axioms act as a *bridge* between them.

When confronted with a situation such as the justifications in Example 17, addressing the derived justification first may lead a user to modify or remove the bridge axioms before the root axioms, thus only affecting the derived entailment. This would require the user to repair the root justification in another step, which would double the effort involved to repair both entailments. This example shows that when attempting to repair multiple entailments, addressing root justifications before derived justifications generally requires the inspection of fewer justifications, thus reducing user effort.

Equality and Inferential Power

Equality is another special case of justification overlap, where the same subset $\mathcal{J}$ of axioms in an ontology $\mathcal{O}$ is a minimal entailing set for several different entailments. We refer to the number of entailments for which $\mathcal{J}$ is a justification as the *inferential power* of $\mathcal{J}$, answering the question 'how much can be expressed with how little?'. It is clear to see that for non-laconic justifications the inferential power depends largely on the size and complexity of the axioms in the justifications; we can easily construct an axiom with high inferential power by creating a large conjunction of subexpressions, where only some of the subexpressions are relevant for a given entailment. On the other hand, this is not the case for laconic justifications, as these will not contain any superfluous information with respect to an entailment.

For justification equality, the j-graph representation has clear benefits over a representation of multiple justifications as lists of axiom sets. Several justifications for

multiple entailments which are equivalent are simply represented by a single justification vertex; the inferential power of a justification corresponds to its out-degree in the j-graph.

4.3 J-Graph Computation

4.3.1 Graph Generation

Generating a j-graph to represent the justificatory structure of an OWL ontology is a fairly straightforward process. First, all axioms in the union of the ontology and the selected entailment set are labelled with a unique identifier a_i. We then compute the justifications for the selected entailments; the justifications are also labelled with unique identifiers j_k. The axiom labelling can also be performed on an existing set of justifications, which has no effect on the general structure of the process.

The j-graph is then constructed as follows: Generate a vertex for each axiom in ax+entset (each justification in J) and label it with the respective identifier. For each justification vertex labelled j_k, generate an edge (a_i, j_k) for each axiom a_i; this will create the edges representing the relation 'axiom occurs in justification'. For each entailment vertex a_i with a justification j_k, generate an edge (j_k, a_i) which represents the relation 'justification for entailment'. This construction requires only a single pass over all given justifications, axioms, and entailments.

4.3.2 Implementation

The above algorithm has been implemented using the OWL API version 3.2.4, and the JGraphT[2] version 0.8.3 graph library for representing the j-graph. Construction of the graph on a set of existing justifications and entailments is performed quickly, with the average time to construct a graph being less than 10 seconds in a set of ontologies ranging from 2 to approximately 11,000 justifications.

While the overlap detection could have been implemented using a naive algorithm which checks for shared axiom sets, we used an existing algorithm from the area of Formal Concept Analysis (FCA). [3] This 'next concept' algorithm (often simply referred to as 'Ganter's algorithm') provides an efficient means for computing the largest shared axiom sets between justifications. The ToscanaJ FCA framework[4] includes a straightforward implementation of Ganter's algorithm, which we used for overlap detection. The justifications correspond to the *objects* in the context, and the axioms correspond to the *attributes*.

[2] http://jgrapht.org/
[3] More on FCA here?
[4] `http://toscanaj.sourceforge.net/`

4.4 Summary and Conclusions

This chapter introduced the notion of the justificatory structure of OWL ontologies, defined a graph-based presentation of the justificatory structure, and gave a brief overview of the implementation of a generator for such graphs. We discussed the occurrence of multiple justifications in OWL ontologies, and the various properties associated with entailments and their justifications. Some of these measurements, such as the numbers and sizes of justifications for a finite entailment set of an ontology, or the activity of an ontology, that is, the proportion of ontology axioms that occur in justifications for an entailment set, give us insight into the implicit structure of an ontology, thus extending the standard metrics used to describe OWL ontologies.

Other structural aspects are based on relations between multiple justifications, such as shared axioms (overlap) between justifications. Justifications for both single and multiple entailments may share some axioms, i.e. the justifications *overlap* to a certain extent. Justification overlaps are key in finding good repairs for a set of justifications, as they may lead to a smaller repair (i.e. hitting set) over the justifications. Furthermore, overlapping justification subsets may also support users in *understanding* multiple justifications by generating lemmas, that is, intermediate entailments, which summarise a set recurring set of axioms. In the case of multiple entailments relations such as root and derived and justification equivalence even seem crucial to reducing user effort for justification understanding and repair, as they highlight the actual (number of) reasons for the entailments. An in-depth discussion of how justificatory structure can be exploited in the debugging process will follow in Chapter 6; for now, we will continue our exploration of justificatory structure by focussing on one particular structural property, namely equivalence relations over justifications in the form of different isomorphisms.