A Novel Approach to Ontology Classification☆

Birte Glimm[a], Ian Horrocks[b], Boris Motik[b], Rob Shearer[b], Giorgos Stoilos[b,*]

[a] *Ulm University, Institute of Artificial Intelligence, 89069 Ulm, DE*
[b] *University of Oxford, Department of Computer Science, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK*

Abstract

Ontology classification—the computation of the subsumption hierarchies for classes and properties—is a core reasoning service provided by all OWL reasoners known to us. A popular algorithm for computing the class hierarchy is the so-called Enhanced Traversal (ET) algorithm. In this paper we present a new classification algorithm that attempts to address certain shortcomings of ET and improve its performance. Apart from classification of classes, we also consider object and data property classification. Using several simple examples, we show that the algorithms commonly used to implement these tasks are incomplete even for relatively weak ontology languages. Furthermore, we show that property classification can be reduced to class classification, which allows us to classify properties using our optimised algorithm. We implemented all our algorithms in the OWL reasoner HermiT. The results of our performance evaluation show significant performance improvements on several well-known ontologies.

Keywords: Ontologies, OWL, Class Classification, Property Classification, Optimisations

1. Introduction

Ontologies expressed using the Web Ontology Language (OWL) and its revision OWL 2 [3, 4] play a central role in the development of the Semantic Web. They are also widely used in biomedical information systems [5, 6, 7], as well as an increasing range of application domains such as agriculture [8], astronomy [9], defence [10], and geography [11]. Ontology classification—the computation of the subsumption hierarchies for classes and properties—is a core reasoning service provided by all OWL reasoners known to us. The resulting class and property hierarchies are used by ontology engineers to navigate the ontology and identify modelling errors, as well as for inference, explanation, and query answering.

Most OWL reasoners, such as Pellet [12], FaCT++ [13], and RacerPro [14], solve the classification problem using an Enhanced Traversal (ET) classification algorithm similar to the one used in early description logic reasoners [15]. To construct a class hierarchy, this algorithm starts with the empty hierarchy and then iteratively inserts each class from the ontology into the hierarchy. Each insertion step typically requires one or more subsumption tests—checks whether a subsumption relationship holds between two classes—in order to determine the proper position of a class in the hierarchy constructed thus far. Significant attention has been devoted to the optimisation of individual subsumption tests [16, 17, 18, 19, 20, 21]. Nevertheless, the ET algorithm can be inefficient on ontologies with a large number of classes: even if each subsumption test is very efficient, the large number of tests required to construct a hierarchy can make classification an expensive operation. Furthermore, repeatedly traversing a large class hierarchy during each insertion step can be costly; this is particularly acute on the relatively flat (i.e., broad and shallow) tree-shaped hierarchies often found in manually constructed ontologies. In order to overcome these deficiencies, algorithms for efficient classification of OWL 2 EL [22] and OWL 2 QL ontologies [23] have been proposed; however, it is currently not known how to apply these algorithms to OWL 2 DL ontologies.

Motivated by the desire to improve the performance of class classification, in this paper we present a novel classification algorithm that can greatly reduce the number of required subsumption tests. Unlike ET, our algorithm does not construct the hierarchy directly; instead, it maintains the sets of known (K) and remaining possible (P) subsumer pairs, and it performs subsumption tests to augment K and reduce P until K contains all the relevant subsumptions and P becomes empty. Such a representation of the hierarchy allows one to manipulate K and P using highly-tuned algorithms, such as the ones for computing the transitive closure and the transitive reduction of a relation. Furthermore, the relatively small subset of P that contains the remaining possible subsumers of a single class can be efficiently extracted using simple operations, which can greatly reduce the cost of hierarchy traversal. To further reduce the number of subsumption tests, we exploit the transitivity of the subclass relation to propagate (non-)subsumption information and thus speed up the process of augmenting K and reducing P.

☆This is a revised and extended version of the work presented in [1, 2].

*Corresonding Author

Email addresses: birte.glimm@uni-ulm.de (Birte Glimm), ian.horrocks@cs.ox.ac.uk (Ian Horrocks), boris.motik@cs.ox.ac.uk (Boris Motik), rob.shearer@cs.ox.ac.uk (Rob Shearer), giorgos.stoilos@cs.ox.ac.uk (Giorgos Stoilos)

The practicability of such an algorithm critically depends on several factors. The first question is how to initialise K and P. We have developed an initialisation approach that exploits information from reasoning tests in order to eagerly identify subsumption relations and unsatisfiable classes and thus reduce the overall amount of work. The second question is how to propagate (non-)subsumptions in K and P efficiently: a naïve strategy, such as the one from [1], can be very inefficient in practice. We have addressed this problem by again exploiting information gathered during reasoning tests.

Apart from the classification of classes, we also consider the classification of object and data properties. To the best of our knowledge, all state of the art OWL reasoners construct property hierarchies by simply computing the reflexive–transitive closure of the subproperty axioms in the ontology. Such a procedure is incomplete, as can be demonstrated by a simple example that uses existential restrictions (ObjectSomeValuesFrom), functional properties, and property hierarchies (i.e., the example can be expressed in OWL Lite), or an example that uses existential restrictions, property chains (ObjectPropertyChain), and inverse properties. Surprisingly, however, the problem of efficiently and correctly constructing property hierarchies has received almost no attention in the literature, even though this is a standard reasoning task extensively used by ontology editors such as Protégé. Property classification can in theory be solved using an algorithm such as ET; however, as we discuss in more detail, such an approach is unlikely to be efficient. As a remedy, we present a novel encoding of the property classification problems into class classification problems, which allows us to exploit our new class classification algorithm to correctly and efficiently compute property hierarchies.

We have implemented our techniques in the HermiT reasoner. To the best of our knowledge, this makes HermiT the only OWL 2 DL reasoner that correctly classifies object and data properties. Moreover, we have conducted an extensive experimental evaluation, which shows that our algorithms consistently outperform ET, sometimes by a factor of ten or more.

2. Preliminaries

In this section we briefly introduce OWL [3, 4]—the ontology language developed by the W3C; we present an overview of the Enhanced Traversal (ET) algorithm [15]; and we present an overview of the model-building calculi, such as tableau and hypertableau, that provide the algorithmic foundation for subsumption checking in most state of the art OWL reasoners.

2.1. OWL Ontologies

In this paper we focus on OWL 2 ontologies interpreted under the Direct Semantics; however, our techniques are also applicable to OWL, as well as any propositionally closed ontology language. For a full definition of OWL 2, please refer to the OWL 2 Structural Specification and Direct Semantics [3, 4]; here we just recapitulate the relevant terminology. A domain of interest can be modelled in OWL 2 by means of *individuals* (which denote objects from the domain of discourse), *literals* (which denote data values, such as strings or integers), *classes* (which denote sets of individuals), *datatypes* (which denote sets of data values), *object properties* (which relate pairs of individuals), and *data properties* (which relate individuals with concrete values). Individuals, classes, datatypes, and object properties can be used to form *class expressions*, *data ranges*, and *object property expressions*, respectively; these are complex descriptions of sets of individuals, sets of literals, and relationships between individuals. Finally, class expressions, data ranges, object property expressions, data properties, individuals, and literals can be used to form *axioms*—statements that describe the domain being modelled. Axioms describing individuals are commonly called *assertions*. An OWL 2 *ontology* O is a finite set of axioms.

For example, consider axioms (1)–(4) below.[1] Axiom (1) states that the class Human is a subclass of the class Animal (i.e., all humans are animals). Axiom (2) states that the individual Alex is an instance of the class Human, while axiom (3) states that the individual Alex is related to literal "27"^^xsd:integer by the data property hasAge (i.e., the age of Alex is 27). Finally, axiom (4) states that the value of the object property hasColour must be an instance of the class Colour.

SubClassOf(Human Animal) (1)

ClassAssertion(Human Alex) (2)

DataPropertyAssertion(hasAge Alex "27"^^xsd:integer) (3)

ObjectPropertyRange(hasColour Colour) (4)

The semantics of axioms in an OWL ontology O is given by means of two-sorted interpretations over the *object domain* and the *data domain*, where the latter contains well-known data values such as integers and strings. An *interpretation* I maps individuals to elements of the object domain, literals to elements of the data domain, classes to subsets of the object domain, datatypes to subsets of the data domain, object properties to sets of pairs of object domain elements, and data properties to sets of pairs whose first component is from the object domain and whose second component is from the data domain. OWL 2 contains two classes, one datatype, two object properties, and two data properties which are all interpreted in every interpretation in a predetermined way. In particular, class owl:Thing is mapped to the set of all objects in the object domain, and class owl:Nothing is mapped to the empty set. Similarly, datatype rdfs:Literal is mapped to the set of all data values in the data domain. Furthermore, object property owl:topObjectProperty is mapped to the set of all pairs of objects from the object domain, and object property owl:bottomObjectProperty is mapped to the empty set. Finally, data property owl:topDataProperty is mapped to all pairs consisting of an object from the object domain and an object from the data domain, and data property owl:bottomDataProperty is mapped to the empty set. An individual i is an *instance* of a class C in an interpretation I if the image of C contains the image of i. For an object property op, an individual i is an *op-successor* of an individual j in an inter-

[1] All elements in OWL are identified using IRIs, but for brevity we do not use IRIs and prefix names in this paper.

pretation I if the image of op contains $\langle \alpha, \beta \rangle$, where α and β are the images of i and j, respectively.

An interpretation I is a *model* of an ontology O if I satisfies all conditions listed in [4]. For example, if O contains axiom (5), then the conditions from [4] require each instance of C in I to be an instance of D in I. As another example, if O contains axiom (6), then i must have an op-successor j in I that is an instance of C in I.

$$\text{SubClassOf}(C\ D) \tag{5}$$

$$\text{ClassAssertion}(\text{ObjectSomeValuesFrom}(op\ C)\ i) \tag{6}$$

If the axioms of O cannot be satisfied in any interpretation (i.e., if O has no model), then O is *unsatisfiable*; otherwise, O is *satisfiable*. If the interpretation of a class C is contained in the interpretation of a class D in all models of O, then C is a *subclass* of D (or, equivalently, D *subsumes* C) in O and we write $O \models C \sqsubseteq D$. If the interpretations of C and D necessarily coincide in all models of O, then C and D are *equivalent* in O and we write $O \models C \equiv D$. A class C is *satisfiable* if a model of O exists in which the interpretation of C is not empty; otherwise, C is *unsatisfiable*. If $O \not\models C \sqsubseteq D$, then a model I of O exists in which C has an instance that is not an instance of D. We use analogous notation for object and data properties.

We use the following notation for sets of entities occurring in an ontology O:

- $\mathbf{C}'_O$ is the set of all classes that occur in O different from owl:Thing and owl:Nothing;
- $\mathbf{OPE}'_O$ contains op and ObjectInverseOf(op) for each object property op that occurs in O and that is different from owl:topObjectProperty and owl:bottomObjectProperty;
- $\mathbf{DP}'_O$ contains each data property that occurs in O different from owl:topDataProperty and owl:bottomDataProperty.

Furthermore, we use the following abbreviations as well:

$$\begin{aligned}
\mathbf{C}_O &= \mathbf{C}'_O \cup \{\text{owl:Thing}, \text{owl:Nothing}\} \\
\mathbf{OPE}_O &= \mathbf{OPE}'_O \cup \{\text{owl:topObjectProperty}, \\
&\qquad\qquad \text{owl:bottomObjectProperty}\} \\
\mathbf{DP}_O &= \mathbf{DP}'_O \cup \{\text{owl:topDataProperty}, \\
&\qquad\qquad \text{owl:bottomDataProperty}\}
\end{aligned}$$

We next illustrate these definitions by means of an example. Let O be the ontology containing axioms (7) and (8); then, $O \models C \sqsubseteq E$ even though this is not stated explicitly. This is because axiom (7) ensures that, in each model of O, each instance i of C is related to an instance of D using the object property op. Each i thus has an op-successor, so the property domain axiom (8) ensures that i is also an instance of E. Since this holds for an arbitrary i, we can conclude that C is a subclass of E.

$$\text{SubClassOf}(C\ \text{ObjectSomeValuesFrom}(\text{op}\ D)) \tag{7}$$

$$\text{ObjectPropertyDomain}(\text{op}\ E) \tag{8}$$

2.2. Enhanced Traversal Algorithm

Classification of an ontology O is the computation of all pairs of classes $\langle C, D \rangle$ such that $\{C, D\} \subseteq \mathbf{C}_O$ and $O \models C \sqsubseteq D$; similarly, object (resp. data) property classification of O is the computation of all pairs of object property expressions (resp. data properties) $\langle R, S \rangle$ such that $\{R, S\} \subseteq \mathbf{OPE}_O$ (resp. $\{R, S\} \subseteq \mathbf{DP}_O$) and $O \models R \sqsubseteq S$. Roughly speaking, for a relation U containing all the resulting pairs, the corresponding hierarchy is the reflexively and transitively reduced relation H that 'implies' all pairs in U.[2] For example, from an ontology that contains (7) and (8), a classification algorithm should compute the following hierarchy:

$$\{ \langle \text{owl:Nothing}, C \rangle, \langle \text{owl:Nothing}, D \rangle, \\ \langle C, E \rangle, \langle E, \text{owl:Thing} \rangle, \langle D, \text{owl:Thing} \rangle \}$$

A naïve way to classify O is to check whether $O \models C \sqsubseteq D$ holds for all possible pairs of C and D in O. Given n classes, such an algorithm requires n^2 tests, which is inefficient even on medium-sized ontologies. To obtain a practical classification algorithm, numerous optimisations have been developed with the goal of reducing the number of tests performed. A prominent such technique is the Enhanced Traversal (ET) algorithm [15]. The algorithm starts with the trivial hierarchy $H = \{\langle \text{owl:Nothing}, \text{owl:Thing} \rangle\}$ and it progressively adds new classes to H using a two-phase procedure. In the first phase, the most specific superclasses of a class C are determined using a top-down breadth-first traversal of H; in the second-phase, the most general subclasses of C are determined using a bottom-up traversal of H.

A sample run of the ET algorithm on an ontology O containing axioms (7) and (8) is shown in Figure 1. The algorithm starts by setting $H = \{\langle \text{owl:Nothing}, \text{owl:Thing} \rangle\}$. Next, the algorithm inserts C into H using the following two steps:

- In the top-down phase, the algorithm checks whether subsumption $O \models C \sqsubseteq \text{owl:Thing}$ holds. This is trivially the case, so the algorithm proceeds with the 'children' of owl:Thing; so far, this includes only owl:Nothing, so the algorithm checks $O \models C \sqsubseteq \text{owl:Nothing}$, which does not hold. Consequently, C must be inserted into H somewhere between owl:Thing and owl:Nothing.
- In the bottom-up phase, the algorithm checks whether subsumption $O \models \text{owl:Nothing} \sqsubseteq C$ holds. This is trivially the case, so the algorithm next checks $O \models \text{owl:Thing} \sqsubseteq C$. The latter subsumption does not hold, so C is inserted into H exactly between owl:Nothing and owl:Thing.

In an analogous way, D is next inserted into H exactly between owl:Nothing and owl:Thing, but in a separate branch from C since $O \not\models C \sqsubseteq D$ and $O \not\models D \sqsubseteq C$. Finally, since $O \models C \sqsubseteq E$, $O \not\models E \sqsubseteq C$, and $O \not\models D \sqsubseteq E$, class E is inserted into H below owl:Thing and above C.

[2] The reflexive–transitive reduction of a binary relation R is the minimal relation R' such that the reflexive–transitive closure of R' is the same as the reflexive–transitive closure of R.

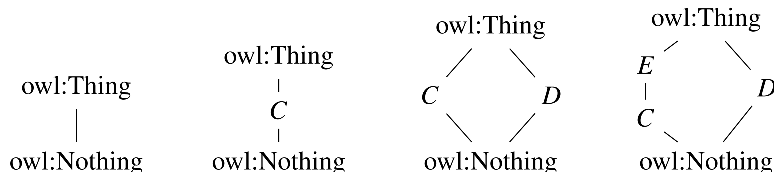


Figure 1: A run of ET over O containing axioms (7) and (8)

The ET algorithm significantly reduces the number of subsumption tests from the theoretical upper bound of n^2. For example, in the top-down phase, if $O \not\models C \sqsubseteq D$, then the algorithm does not need to check C against the children of D. Nevertheless, classifying large ontologies might still require a large number of subsumption tests. This is because most real-world ontologies usually have relatively flat (i.e., broad and shallow) hierarchies with only a few 'top-level' classes (i.e., classes located immediately below owl:Thing in the hierarchy). In such cases, most classes have owl:Nothing as a child, so in the bottom-up phase one must check the subsumption of a class against a (possibly) large number of such 'leaf' classes. Furthermore, as H becomes larger in size, repeated traversal of H in both the top-down and bottom-up phases can be costly.

In order to further reduce the number of subsumption tests required to compute the hierarchy, additional optimisation techniques have been proposed. Most of these try to identify obvious (non-)subsumptions by propagating information from previous tests [15] or via cheap syntactic checks, such as *told subsumers* [15], *told non-subsumers* [24], and *completely defined classes* [25]. While such optimisations can significantly improve the performance of the ET algorithm, they do not overcome all the problems outlined above.

2.3. Model Construction using (Hyper)Tableau Calculi

It is well known that checking subsumption between classes C and D w.r.t. an ontology O (i.e., checking if $O \models C \sqsubseteq D$) is equivalent to checking whether the class

$$A = \text{ObjectIntersectionOf}(C\ \text{ObjectComplementOf}(\ D\ ))$$

is unsatisfiable w.r.t. O, which is equivalent to checking whether

$$O \cup \{\text{ClassAssertion}(A\ s_0)\}$$

is unsatisfiable for s_0 a 'fresh' individual (i.e., an individual not occurring in O). To decide the latter problem, most OWL reasoners use a model construction calculus, such as tableau or hypertableau. Please refer to [26] for a detailed introduction to the hypertableau calculus for OWL 2, and to [27] for the tableau calculus; here, we just present an overview of the aspects of these calculi that are relevant to our classification algorithm.

Although (hyper)tableau calculi have been formalised in a variety of ways, all of them can be seen as constructing a *generalised* set of assertions that represents (an abstraction of) a model of O. Each such calculus consists of one or more derivation rules that can be applied to a set of assertions $\mathcal{A}$ to produce a set of assertions $\mathcal{A}'$, where the latter set makes a certain piece of information from O explicit. Derivation rules usually add new class or property assertions, and they may introduce new individuals; the latter may be necessary to satisfy, for example, existential restrictions (ObjectSomeValuesFrom). Moreover, in addition to standard assertions, derivation rules can add a special assertion unsatisfiable if an obvious contradiction is detected. Finally, derivation rules can be nondeterministic—that is, a derivation rule can be allowed to choose between several alternative assertions to add. To show that A is satisfiable, (hyper)tableau calculi construct a *derivation* for O and A—a sequence of sets of assertions $\mathcal{A}_0, \ldots, \mathcal{A}_n$ where

- $\mathcal{A}_0$ contains all assertions in O as well as the assertion ClassAssertion(A s_0), where s_0 is a fresh individual called the *root*,
- $\mathcal{A}_{i+1}$ is a *possible* result of applying a derivation rule to $\mathcal{A}_i$ for each $0 < i \leq n$, and
- no derivation rule is applicable to $\mathcal{A}_n$.

If a derivation for O and A exists such that $\mathcal{A}_n$ does not contain unsatisfiable, then A is satisfiable and $\mathcal{A}_n$ is called a *pre-model* for A. If no such derivation exists, then A is unsatisfiable (i.e., it is equivalent to owl:Nothing).

Each assertion occurring in a derivation $\mathcal{A}_0, \ldots, \mathcal{A}_n$ is derived either *deterministically* or *nondeterministically*, which is determined inductively as follows: all assertions in $\mathcal{A}_0$ are derived deterministically; furthermore, an assertion occurring in some $\mathcal{A}_i$ is derived deterministically if and only if it is derived using a deterministic derivation rule from assertions that were all derived deterministically. In the rest of this paper we assume that we can determine for each assertion α occurring in some $\mathcal{A}_i$ how α was derived. This is straightforward in practice since all state of the art (hyper)tableau reasoners employ *dependency directed backtracking* [17]. In order to optimise backtracking, these reasoners associate with each assertion α a *dependency set*—a data structure that indicates the nondeterministic choices that α depends on. Then, α is derived deterministically if and only if the dependency set of α is empty. Discussing the details of dependency directed backtracking is out of scope of this paper; please refer to [17] for further details.

For a set of assertions $\mathcal{A}$ and individuals s and t that appear in $\mathcal{A}$, we define the *label* $\mathcal{L}_{\mathcal{A}}(s)$ of s in $\mathcal{A}$ as follows:

$$\mathcal{L}_{\mathcal{A}}(s) := \{A \mid \text{ClassAssertion}(A\ s) \in \mathcal{A} \text{ and } A \text{ is a class}\}$$

The classification algorithm presented in this paper can be used with any (hyper)tableau calculus for which each pre-model $\mathcal{A}_n$ for O and A with root individual s_0 produced by the calculus satisfies the following property:

(P1) if $C \in \mathcal{L}_{\mathcal{A}_n}(s_0)$ and the assertion ClassAssertion(C s_0) was derived deterministically, then $O \models A \sqsubseteq C$.

All (hyper)tableau calculi used in practice that we are aware of satisfy this property and so they can be used with our classification algorithm.

In addition, for each ontology O and each pre-model $\mathcal{A}_n$ generated by the hypertableau calculus used in the HermiT reasoner [26], the following property holds:

(P2) for an *arbitrary* individual s in $\mathcal{A}_n$ and arbitrary classes D and E, if $D \in \mathcal{L}_{\mathcal{A}_n}(s)$ and $E \notin \mathcal{L}_{\mathcal{A}_n}(s)$, then $O \not\models D \sqsubseteq E$.

Pre-models produced by tableau algorithms as presented in the literature also satisfy property (P2); however, commonly used optimisations, such as lazy unfolding [15], can compromise property (P2). Nevertheless, most (if not all) implemented calculi produce pre-models that satisfy at least the following weaker property:

(P3) for an *arbitrary* individual s in $\mathcal{A}_n$ and arbitrary classes D and E where E is primitive in O,[3] if $D \in \mathcal{L}_{\mathcal{A}_n}(s)$ and $E \notin \mathcal{L}_{\mathcal{A}_n}(s)$, then $O \not\models D \sqsubseteq E$.

As the following example shows, properties (P2) and (P3) can be used to extract (non-)subsumptions from pre-models.

Example 1. Let O be an ontology that contains the following axioms:

$$\text{SubClassOf}(A\ B) \tag{9}$$

$$\text{SubClassOf}(B\ C) \tag{10}$$

$$\text{SubClassOf}(E\ F) \tag{11}$$

To check whether A is satisfiable, a (hyper)tableau calculus constructs a pre-model that satisfies properties (P1) and (P3). In particular, the calculus starts with the set of assertions $\mathcal{A}_0 = \{$ ClassAssertion(A s_0) $\}$. To satisfy the axioms in O, the calculus extends $\mathcal{A}_0$ with ClassAssertion(B s_0) and ClassAssertion(C s_0); let $\mathcal{A}_n$ be the resulting pre-model. All practical (hyper)tableau calculi we are aware of are sufficiently optimised so as to produce $\mathcal{A}_n$ deterministically. We can now use the label $\mathcal{L}_{\mathcal{A}_n}(s_0)$ to identify (non-)subsumers of A as follows. Since E and F are primitive in O, from $E \notin \mathcal{L}_{\mathcal{A}_n}(s_0)$ and $F \notin \mathcal{L}_{\mathcal{A}_n}(s_0)$ we can conclude that neither E nor F is a subsumer of A; this is because $\mathcal{A}_n$ is an abstraction of a model of O that witnesses the non-subsumption. Furthermore, from the fact that all assertions in $\mathcal{A}_n$ were derived deterministically, we can conclude that B and C are subsumers of A. ◇

3. Optimised Class Classification

In this section we introduce our classification algorithm. We discuss the main ideas and present an overview of the algorithm in Section 3.1, after which we present the algorithm in full detail in Sections 3.2 and 3.3.

[3] A class E is said to be *primitive* in O if O is unfoldable [25] and it does not contain an axiom of the form EquivalentClasses(E C).

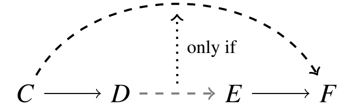


Figure 2: Eliminating impossible relationships: nodes represent classes, solid edges represent pairs in K, and the grey edge represents a pair that can be in P only if the pair represented by the dashed edge is in $P \cup K$.

3.1. An Overview

In order to reuse the (non-)subsumption information from satisfiability and subsumption tests, our algorithm maintains two binary relations on $\mathbf{C}_O \times \mathbf{C}_O$ which we denote with K and P. Relation K represents the known subsumptions—that is, $\langle C, D\rangle \in K$ implies that $O \models C \sqsubseteq D$ is known for certain. One might be tempted to use a dual relation that represents the known non-subsumptions; however, such a relation is typically quite large, so maintaining it explicitly would be impractical. Our algorithm therefore manages the non-subsumption information indirectly using a relation P of *possible* subsumptions. More precisely, $\langle C, D\rangle \in P$ and $\langle C, D\rangle \notin K$ implies that $O \models C \sqsubseteq D$ is possible (i.e., no evidence to the contrary has been encountered thus far); thus, $\langle C, D\rangle \notin P$ and $\langle C, D\rangle \notin K$ imply that $O \not\models C \sqsubseteq D$ is known. Apart from initialisation and during certain intermediate steps, relations K and P are disjoint; thus, P reflects the 'remaining work' needed to classify O.

Given a class C for which another class D exists such that $\langle C, D\rangle \in P$, our algorithm extracts from K and P a partial hierarchy H_C of all unknown, but possible subsumers of C, and it then inserts C into H_C using a variant of the ET algorithm. Class C will typically have many known subsumers but few unknown and possible subsumers, so H_C will usually be small. During the insertion of C into H_C, our algorithm expands K and prunes P using the information obtained in subsumption tests, thus potentially reducing the work needed to classify the remaining classes. This process is repeated until P becomes empty, at which point the transitive closure of K precisely captures the subsumption relation between classes in O.

Our algorithm systematically exploits the transitivity of $\sqsubseteq$ to extend K and prune P without actual reasoning. For example, if $\{\langle C, D\rangle, \langle E, F\rangle\} \subseteq K$ and a subsumption test requires adding $\langle D, E\rangle$ to K, then $\langle C, F\rangle$ can be added to K as well due to the transitivity of $\sqsubseteq$. Ideally, our algorithm would maintain the transitive closure of K as new subsumptions are discovered. Efficient algorithms for the maintenance of transitive closures under updates are available in the literature; however, we found them to be memory inefficient, which causes problems on large ontologies with many classes, such as FMA and SNOMED. Therefore, instead of computing the transitive closure of K explicitly, our algorithm uses a graph reachability algorithm to identify whether a pair of the form $\langle C, D\rangle$ belongs to the transitive closure of K.

The transitivity of the subsumption relation can also be used to remove obvious non-subsumptions from P. For example, if $\{\langle C, D\rangle, \langle E, F\rangle\} \subseteq K$ and $\{\langle D, E\rangle, \langle C, F\rangle\} \subseteq P$, and we discover

that $\langle C, F\rangle$ should be removed from P (because C is not subsumed by F), then we can remove $\langle D, E\rangle$ from P as well: if $\langle D, E\rangle$ were later added to K, then $\langle C, F\rangle$ would need to be added to K as well due to the transitivity of the subsumption relation, which contradicts our evidence that C is not subsumed by F. Analogously, if $\{\langle C, D\rangle, \langle D, E\rangle\} \subseteq P$, $\langle E, F\rangle \in K$, and $\langle C, F\rangle \notin P \cup K$, and we discover that $\langle C, D\rangle$ should be added to K, then we can remove $\langle D, E\rangle$ from P. Such situations are shown schematically in Figure 2.

Note, however, that checking conditions from the previous paragraph requires several nested loops over potentially very large relations K and P; thus, a direct implementation of such a scheme, as originally suggested in [1], is unlikely to be efficient in practice. Our algorithm therefore uses a different pruning strategy. Assume that a subsumption or a class satisfiability test produces a pre-model $\mathcal{A}$ satisfying property (P2) from Section 2.3. For each individual s in a pre-model $\mathcal{A}$ and each class $D \in \mathcal{L}_{\mathcal{A}}(s)$, if $\langle D, E\rangle \in P$ but $E \notin \mathcal{L}_{\mathcal{A}}(s)$, then we can remove $\langle D, E\rangle$ from P: if E were a subsumer of D, then s would be an instance of E in every pre-model, including $\mathcal{A}$. We present a variant of this scheme that is applicable if $\mathcal{A}$ satisfies only the weaker property (P3). Although such approaches only partially capture the pruning scheme from [1], they seem to exhibit a good balance between efficiency of pruning and reduction of the number of subsumption tests.

Before presenting our algorithm, we next introduce several definitions. For example, we define precisely what a hierarchy is, and we define certain shortcuts for manipulating K and P. In order to use the same definitions for class and property hierarchies, we present the definitions for a general set U containing elements $E_\top$ and $E_\bot$, and a subset S of U. To apply these definitions to class classification, one should take U to be the set of all classes in an ontology, $E_\top$ and $E_\bot$ should be owl:Thing and owl:Nothing, respectively, and S should be the set of classes that we want to classify.

Definition 2. *Let U be a set containing special elements $E_\bot$ and $E_\top$, let S be a subset of U, and let $R \subseteq U \times U$ be a binary relation on U.*

- *For $C \in U$, let $R|_C = \{D \mid \langle C, D\rangle \in R\}$.*
- *For $C, D \in U$, element D is* reachable *in R from element C, written $C \rightsquigarrow_R D$, if elements $E_0, \ldots, E_n \in U$ with $n \geq 0$ exist such that $E_0 = C$, $E_n = D$ and $\langle E_i, E_{i+1}\rangle \in R$ for each $0 \leq i < n$.[4] The opposite of reachable is written $C \not\rightsquigarrow_R D$.*
- *A* hierarchy *of S w.r.t. R, $E_\bot$, and $E_\top$ is a triple (V, H, ρ) whose components satisfy the conditions listed below, for T defined as*

$$T = \{\langle E_\bot, C\rangle, \langle C, E_\top\rangle \mid C \in S\} \cup \{\langle E_\bot, E_\top\rangle\} \cup \{\langle C, D\rangle \in S \times S \mid C \rightsquigarrow_R D\}.$$

  - *V is a set that contains, for each $D \in S \cup \{E_\bot, E_\top\}$, precisely one $C \in S \cup \{E_\bot, E_\top\}$ such that $C \rightsquigarrow_T D$ and $D \rightsquigarrow_T C$.*
  - *H is the reflexive–transitive reduction of the relation $\{\langle C, D\rangle \in V \times V \mid C \rightsquigarrow_T D\}$.*
  - *$\rho : V \to 2^{S \cup \{E_\bot, E_\top\}}$ is the function on V such that $D \in \rho(C)$ if and only if $C \rightsquigarrow_T D$ and $D \rightsquigarrow_T C$.*
- *Function* hierarchy$(S, R, E_\bot, E_\top)$ *returns one arbitrarily chosen but fixed hierarchy of S w.r.t. R, $E_\bot$, and $E_\top$.*

[4]Note that, according to this definition, each $C \in U$ is reachable from itself.

Intuitively, hierarchy$(S, R, E_\bot, E_\top)$ arranges the elements of S into a hierarchy where $E_\bot$ and $E_\top$ are bottom and top elements, respectively, and which 'preserves' the order of R; if S contains a subset of the elements of R, the result can be understood as a 'projection' of R to S. The set V contains a single 'representative' C for each strongly connected component of T, and $\rho(C)$ contains precisely the vertices of the strongly connected component of T that contains C. Note that the result of hierarchy$(S, R, E_\bot, E_\top)$ is unique up to the choice of the representative for each strongly connected component of T; furthermore, relation T contains the reflexive–transitive closure of R, but one does not necessarily need to materialise the closure in order to determine hierarchy$(S, R, E_\bot, E_\top)$; finally, if $S \cup \{E_\bot, E_\top\} = U$ (as is often the case in practice), then the definition of T can be simplified to

$$T = \{\langle E_\bot, C\rangle, \langle C, E_\top\rangle \mid C \in S\} \cup \{\langle E_\bot, E_\top\rangle\} \cup R.$$

Our classification algorithm uses functions buildPreModel, explicitSubsumptions, and possibleSubsumers, which we describe next. These functions should be understood as parameters to our algorithm: one can use arbitrary functions, provided that they satisfy properties specified in Definitions 3–5.

Definition 3. *Let $\mathcal{O}$ be an ontology, and let $\mathcal{D}$ and $\mathcal{N}$ be sets of assertions. Function* buildPreModel$(\mathcal{D}, \mathcal{N}, \mathcal{O})$ *should return a set of assertions that is either a pre-model of $\mathcal{D} \cup \mathcal{N} \cup \mathcal{O}$, or that contains* unsatisfiable *if $\mathcal{D} \cup \mathcal{N} \cup \mathcal{O}$ is unsatisfiable. The result should satisfy property (P1) from Section 2.3, and it should be constructed by treating the assertions in $\mathcal{D}$ and $\mathcal{N}$ as having been derived deterministically and nondeterministically, respectively.*

Function buildPreModel is used to test the satisfiability of a class and subsumption between classes using the (hyper)tableau calculus. In particular, to check the satisfiability of C, our algorithm will call the function with $\mathcal{D}$ and $\mathcal{N}$ defined as follows, where s_0 is a 'fresh' individual:

$$\mathcal{D} = \{\text{ ClassAssertion}(C\ s_0)\ \} \qquad \mathcal{N} = \emptyset$$

If C is satisfiable, the function should return a pre-model of C with root s_0. Furthermore, to check whether $\mathcal{O} \models C \sqsubseteq D$ holds, our algorithm will call the function with $\mathcal{D}$ and $\mathcal{N}$ as follows:

$$\mathcal{D} = \{\text{ ClassAssertion}(C\ s_0)\ \}$$
$$\mathcal{N} = \{\text{ ClassAssertion}(\text{ObjectComplementOf}(D\ s_0))\ \}$$

If the subsumption does not hold, the function should return a pre-model. To understand why buildPreModel accepts as input two distinct sets of assertions $\mathcal{D}$ and $\mathcal{N}$, remember that, as

discussed in Section 2.3, known subsumers for a class A can be identified by performing a satisfiability test for A and then checking which assertions involving the root individual were derived deterministically. We extend this approach in a way that allows us to identify known subsumers during subsumption tests as well. To facilitate this, buildPreModel accepts two sets of assertions. When constructing a pre-model for $\mathcal{D} \cup \mathcal{N} \cup \mathcal{O}$, the assertions in $\mathcal{D}$ are treated as having been derived deterministically, but the assertions in $\mathcal{N}$ are treated as having been derived nondeterministically (in practice, one can achieve this by associating each assertion in $\mathcal{N}$ with a dummy nonempty dependency set). Let $\mathcal{A}$ be the result of applying buildPreModel to $\mathcal{D}$, $\mathcal{N}$, and $\mathcal{O}$; if an assertion $\alpha \in \mathcal{A}$ was derived deterministically, then we know that α was deterministically derived from $\mathcal{D}$ and $\mathcal{O}$ *only*. Thus, when performing a subsumption test $C \sqsubseteq D$, if an assertion ClassAssertion(E s_0) in $\mathcal{A}$ was derived deterministically, then we know for certain that E is a subsumer of C. This is possible even if C is subsumed by D, so $\mathcal{A}$ is not a pre-model. Such a technique extends the approaches in [1, 2] and, as we discuss in Section 6, it significantly improves the performance of classifying the GALEN ontology [7].

Definition 4. *Let S be a set of classes and let $\mathcal{O}$ be an ontology. Function* explicitSubsumptions$(S, \mathcal{O})$ *should return a (possibly empty) set of pairs of classes $\langle C, D\rangle$ such that $C, D \in S$ and $\mathcal{O} \models C \sqsubseteq D$.*

Function explicitSubsumptions$(S, \mathcal{O})$ is used to extract from the ontology $\mathcal{O}$ the 'explicit' class subsumptions—that is, subsumptions that can be extracted from $\mathcal{O}$ using a lightweight, typically syntactic analysis. The result of this function does not need to be transitively closed; in fact, transitively closing the result might adversely affect the performance of the classification algorithm. In the HermiT system, this function returns all pairs of classes $\langle C, D\rangle$ such that $C, D \in S$ and $\mathcal{O}$ contains an axiom of the form SubClassOf(C D) or of the form SubClassOf(C ObjectIntersectionOf(D_1 ... D_n)) in which we have $D = D_i$ for some $1 \le i \le n$.

Definition 5. *Let S be a set of classes, let $\mathcal{O}$ be an ontology, let C be a class, let s be an individual, let $\mathcal{A}$ be a pre-model, and let K be a set of pairs of classes satisfying the following conditions:*

- *$\mathcal{O} \models F_1 \sqsubseteq F_2$ for each $\langle F_1, F_2\rangle \in K$, and*
- *a class E exists such that $E \in \mathcal{L}_{\mathcal{A}}(s)$ and $E \leadsto_K C$.*

Function possibleSubsumers$(S, \mathcal{O}, C, s, \mathcal{A}, K)$ *should return a (not necessarily minimal) subset of S that contains at least each class $D \in S$ such that $\mathcal{O} \models C \sqsubseteq D$.*

Function possibleSubsumers$(S, \mathcal{O}, C, s, \mathcal{A}, K)$ should return a subset of S that contains all candidate subsumers of C; note that this must include C itself. Set K will contain pairs of known subsumers in $\mathcal{O}$—that is, if $F_1 \leadsto_K F_2$ for some $F_1, F_2 \in S$, then $\mathcal{O} \models F_1 \sqsubseteq F_2$. Moreover, the function is called for a class C only if some s and E exists such that $E \in \mathcal{L}_{\mathcal{A}}(s)$ and $E \leadsto_K C$. This condition implies that s should be an instance of C in a model constructed from $\mathcal{A}$, even if $C \notin \mathcal{L}_{\mathcal{A}}(s)$ holds due to various optimisations of the calculus used to construct $\mathcal{A}$. Hence, the function can be called to determine the possible subsumers for classes that do not explicitly appear in $\mathcal{A}$.

In the simplest case, the function can return S; however, one can exploit $\mathcal{L}_{\mathcal{A}}(s)$ and K to return a smaller set of possible subsumers. For example, if $\mathcal{A}$ satisfies property (P2) from Section 2.3, as it is the case in the HermiT system, then we can define possibleSubsumers as

$$\text{possibleSubsumers}(S, \mathcal{O}, C, s, \mathcal{A}, K) = \mathcal{L}_{\mathcal{A}}(s) \cap S.$$

Set K is not useful in this case: if $F \notin \mathcal{L}_{\mathcal{A}}(s)$ and $D \leadsto_K F$, then by property (P2) we have $D \notin \mathcal{L}_{\mathcal{A}}(s)$. In contrast, K is useful if $\mathcal{A}$ satisfies only the weaker property (P3) from Section 2.3: then we can define possibleSubsumers as

$$\begin{aligned}&\text{possibleSubsumers}(S, \mathcal{O}, C, s, \mathcal{A}, K) = \\ &\quad \{D \in S \mid \text{for each primitive class } F \text{ in } \mathcal{O} \\ &\qquad\qquad \text{such that } D \leadsto_K F, \text{ we have } F \in \mathcal{L}_{\mathcal{A}}(s)\}.\end{aligned}$$

The above definition can be intuitively understood as follows. By the conditions on the arguments of possibleSubsumers, we know that $E \in \mathcal{L}(s)$ for some class E such that $E \leadsto_K C$; thus, as explained before, s should be an instance of C in a model constructed from $\mathcal{A}$. Consider now an arbitrary class $D \in S$. If a primitive class F in $\mathcal{O}$ exists such that $F \notin \mathcal{L}_{\mathcal{A}}(s)$, by property (P3) we know that $\mathcal{O} \not\models C \sqsubseteq F$; but then, if $D \leadsto_K F$ holds as well, we clearly have $\mathcal{O} \not\models C \sqsubseteq D$. Such D need not be included in possibleSubsumers$(S, \mathcal{O}, C, s, \mathcal{A}, K)$, and so Definition 5 simply excludes all D that satisfy this condition.

Example 6. Let $\mathcal{O}$ contain axioms (12)–(13).

$$\text{SubClassOf}(A \text{ ObjectSomeValuesFrom}(op \text{ ObjectIntersectionOf}(X\ Y))) \quad (12)$$

$$\text{EquivalentClasses}(B \text{ ObjectSomeValuesFrom}(op\ X)) \quad (13)$$

Clearly, we have $\mathcal{O} \models$ SubClassOf(A B). Furthermore, assume that $K = \{\langle A, A\rangle, \langle B, B\rangle\}$, and that we need to test the satisfiability of A by producing a pre-model $\mathcal{A}$ for A with root s_0.

If $\mathcal{A}$ is produced by a calculus that satisfies property (P2), then $\mathcal{A}$ contains assertion ClassAssertion(B s_0), so we have

$$\text{possibleSubsumers}(S, \mathcal{O}, A, s_0, \mathcal{A}, K) = \mathcal{L}_{\mathcal{A}}(s_0) \cap S = \{A, B\}.$$

Assume now that $\mathcal{A}$ is produced by a calculus that satisfies only property (P3). Then, $\mathcal{A}$ need not contain the assertion ClassAssertion(B s_0) since B is not a primitive class; however, we still have possibleSubsumers$(S, \mathcal{O}, A, s_0, \mathcal{A}, K) = \{A, B\}$. This is because no primitive class is reachable from B in K, so B vacuously satisfies the condition in the definition of possibleSubsumers. Moreover, X and Y are not included into the result of possibleSubsumers$(S, \mathcal{O}, A, s_0, \mathcal{A}, K)$ since both are primitive classes that do not occur in $\mathcal{L}_{\mathcal{A}}(s_0)$. ◇

In addition to the above mentioned three 'parametric' functions, our algorithm uses two fully specified functions called knownSubsumers and prune, which are introduced in Definitions 7 and 8, respectively.

Definition 7. *Let S be a set of classes, let s_0 be an individual, and let $\mathcal{A}$ be a set of assertions. Then, function* knownSubsumers$(S, s_0, \mathcal{A})$ *returns the set containing each $D \in S$ for which $\mathcal{A}$ contains a deterministically derived assertion ClassAssertion$(D\ s_0)$.*

Function knownSubsumers$(S, s_0, \mathcal{A})$ is called in our algorithm with $\mathcal{A}$ a pre-model with root s_0 for a class C. Since $\mathcal{A}$ satisfies property (P1) from Section 2.3, from each deterministically derived assertion ClassAssertion$(D\ s_0)$ in $\mathcal{A}$ we can conclude that $O \models C \sqsubseteq D$.

Definition 8. *Let P and K be sets of pairs of classes, let O be an ontology, and let $\mathcal{A}$ be a set of assertions. Function* prune$(P, O, \mathcal{A}, K)$ *returns the relation obtained from P by removing each pair $\langle C, D\rangle$ for which an individual s in $\mathcal{A}$ and a class E exist such that $E \in \mathcal{L}_{\mathcal{A}}(s)$, $E \rightsquigarrow_K C$, and $D \notin$ possibleSubsumers$(\mathbf{C}_O, O, C, s, \mathcal{A}, K)$.*

Function prune$(P, O, \mathcal{A}, K)$ removes from P certain pairs of classes $\langle C, D\rangle$ for which we have $O \not\models C \sqsubseteq D$. Since P can be very large, this function requires careful implementation in order to obtain an efficient implementation. Assuming that possibleSubsumers$(S, O, C, s, \mathcal{A}, K)$ is defined as outlined after Definition 5, prune$(P, O, \mathcal{A}, K)$ can be efficiently implemented by iteratively considering each individual s in $\mathcal{A}$, each class $E \in \mathcal{L}_{\mathcal{A}}(s)$, each class C such that $E \rightsquigarrow_K C$, and each class $D \in P|_C$; if D does not satisfy the conditions for membership in possibleSubsumers$(S, O, C, s, \mathcal{A}, K)$, then $\langle C, D\rangle$ is removed from P.

Our algorithm for classifying classes is shown in Algorithm 1. The algorithm first checks whether the given ontology is satisfiable; if not, a trivial hierarchy in which all classes are subsumed by owl:Nothing is returned. If the ontology is satisfiable, then the algorithm proceeds with the classification. The goal is to compute a relation K such that, for each class $A \in \mathbf{C}_O$ different from owl:Nothing and each class $B \in \mathbf{C}_O$ different from owl:Thing, we have $O \models A \sqsubseteq B$ if and only if $A \rightsquigarrow_K B$.[5] This is achieved by initialising relations K and P as described Section 3.2, and then processing possible subsumptions in P using a modified version of the ET algorithm as described in Section 3.3. Finally, the algorithm returns a hierarchy derived from K.

3.2. The Initialisation Phase

In [1], relations K and P are initialised by performing a satisfiability test for each class to be added to the hierarchy and then extracting known and possible subsumers from pre-models as discussed in Section 2.3. Although modern reasoners can check satisfiability of classes quite efficiently, the time required to test satisfiability of all the classes can become large if the ontology contains many classes. Moreover, it is likely that many of these satisfiability tests will be redundant and could thus be omitted. For example, if $O \models C \sqsubseteq D$ and C is satisfiable, then a pre-model $\mathcal{A}$ for C also witnesses the satisfiability of D, as well as of any other class occurring in $\mathcal{A}$. We can thus avoid checking the satisfiability of each such D, and we can use $\mathcal{A}$ to extract its possible subsumers. In order to maximise the effect of this optimisation, we start by checking the satisfiability of classes likely to be classified near the bottom of the hierarchy: such classes are likely to produce larger pre-models that are richer in (non-)subsumption information and that thus witness the satisfiability of numerous other classes. The drawback of testing only classes near the bottom of the hierarchy is that we do not determine any known subsumers for classes that are higher in the hierarchy and whose satisfiability can be demonstrated only indirectly. Our approach, however, seems to be quite effective in practice because a substantial number of classes in an ontology end up near the bottom of the hierarchy; furthermore, our strategies for updating K often infer the known subsumptions that are missed in the initialisation phase.

Algorithm 1 Classify(O)

Input: an ontology O whose set of classes $\mathbf{C}_O$ should be classified

1: $\mathcal{A}$:= buildPreModel$(\emptyset, \emptyset, O)$
2: **if** unsatisfiable $\in \mathcal{A}$ **then**
3:     **return** the trivial hierarchy in which each class $C \in \mathbf{C}_O$ is subsumed by owl:Nothing
4: **end if**
5: (K, P) := initialiseRelations$(O, \mathbf{C}_O)$
6: processRemainingClasses$(K, P, O, \mathbf{C}_O)$
7: **return** hierarchy$(\mathbf{C}_O, K, \text{owl:Nothing}, \text{owl:Thing})$

These ideas are captured in Algorithm 2. The algorithm takes as input an ontology O and a set of classes $S \subseteq \mathbf{C}_O$ to be classified. This generality will allow us to reuse the algorithm for the classification of object and data properties with only minor changes (see Sections 4 and 5).

First, K is initialised to all pairs of classes $\langle C, D\rangle$ for which the subsumption is either explicitly stated in O, or whose subsumption can be derived by lightweight transformations of the axioms in O (line 1). Relation K is next used to extract a hierarchy H (line 2), which in many practical cases provides a good approximation of the final hierarchy; this approximate hierarchy H is used next to optimise the order in which the algorithm processes classes.[6] Note also that the algorithm will only process the selected ‘representative’ classes, but this will indirectly classify all classes that are equivalent to the representative. For efficiency reasons, P is initialised in line 3 to $\emptyset$ rather than all possible pairs of classes in S; thus, during the initialisation, $P|_D = \emptyset$ means ‘the possible subsumers of D have not been determined yet’ rather than ‘there are no possible subsumers of D’. Then, set ToTest is initialised (line 4) to contain the leaves of H (i.e., the classes directly above owl:Nothing). A class C is then iteratively removed from ToTest (lines 5–28) and the satisfiability of C is checked (line 8), unless the possible subsumers of C were already determined, or C was determined to be unsatisfiable (line 7). If C is unsatisfiable (lines 10–16), then each D that reaches C in H is unsatisfiable as well (recall that C is reachable from itself); hence, this is recorded in K (line 11), D

[5] Note that the excluded cases of owl:Nothing and owl:Thing are all tautologies, whose management during classification can only add unnecessary overhead.

[6] Note that V and ρ are not used in the rest of the initialisation algorithm.

Algorithm 2 initialiseRelations(O, S)

Input: an ontology O and a set S of classes to be classified

1: K := explicitSubsumptions(S, O)
2: (V, H, ρ) := hierarchy(S, K, owl:Nothing, owl:Thing)
3: $P := \emptyset$
4: ToTest := $\{C \mid \langle \text{owl:Nothing}, C\rangle \in H\}$
5: **while** ToTest $\neq \emptyset$ **do**
6: choose and remove C from ToTest
7: **if** $P|_C = \emptyset$ and $\langle C, \text{owl:Nothing}\rangle \notin K$ **then**
8: $\mathcal{A}$:= buildPreModel($\{\text{ClassAssertion}(C\ s_0)\}, \emptyset, O$) // s_0 is fresh
9: **if** unsatisfiable $\in \mathcal{A}$ **then** // C is unsatisfiable
10: **for all** D such that $D \rightsquigarrow_H C$ and $\langle D, \text{owl:Nothing}\rangle \notin K$ **do**
11: add $\langle D, \text{owl:Nothing}\rangle$ to K
12: remove D from ToTest
13: **for all** $\langle D, E\rangle \in H$ with $\langle E, \text{owl:Nothing}\rangle \notin K$ **do**
14: add E to ToTest
15: **end for**
16: **end for**
17: **else**
18: add $\langle C, D\rangle$ to K for each $D \in$ knownSubsumers($S, s_0, \mathcal{A}$)
19: **for all** s in $\mathcal{A}$, all $D \in \mathcal{L}_{\mathcal{A}}(s)$, and all E such that $D \rightsquigarrow_H E$ **do**
20: **if** $P|_E = \emptyset$ **then**
21: add $\langle E, F\rangle$ to P for each $F \in$ possibleSubsumers($S, O, E, s, \mathcal{A}, K$)
22: **else**
23: remove each $\langle E, F\rangle$ from P such that $F \notin$ possibleSubsumers($S, O, E, s, \mathcal{A}, K$)
24: **end if**
25: **end for**
26: **end if**
27: **end if**
28: **end while**
29: remove each $\langle E_1, E_2\rangle$ from P such that $E_1 \rightsquigarrow_K E_2$
30: **return** (K, P)

is removed from ToTest (line 12), and each parent E of D that is not known to be unsatisfiable is added to ToTest (lines 13–15). In contrast, if C is satisfiable in a pre-model $\mathcal{A}$ with root s_0 (lines 18–25), then the pre-model $\mathcal{A}$ is used to identify the known subsumers of C (line 18). Furthermore, for each class E for which class D and individual s exist such that $D \in \mathcal{L}_{\mathcal{A}}(s)$ and $D \rightsquigarrow_H E$, set P is updated with the possible subsumers of E (lines 19–25). If $P|_E = \emptyset$, this means that the possible subsumers of E have not been initialised; therefore, P is modified to ensure that $P|_E$ contains each class F that is a possible subsumer of E (line 21). If $P|_E \neq \emptyset$, then P is modified by removing those pairs $\langle E, F\rangle$ such that F is not a possible subsumer of E (line 23). Finally, after all possible subsumers of each class in S have been determined, each subsumption that holds according to the information in K can be removed from P (line 29). This now leaves P to reflect the remaining work needed to classify the elements in S.

Note that, if the pre-model $\mathcal{A}$ constructed in line 8 satisfies property (P2) from Section 2.3, then the condition in line 19 can be simplified to consider each s in $\mathcal{A}$ and each class E such that $E \in \mathcal{L}_{\mathcal{A}}(s)$. This is because, if $D \in \mathcal{L}_{\mathcal{A}}(s)$ and $D \rightsquigarrow_H E$ for some class D, by the definition of H we have $O \models D \sqsubseteq E$, and so by property (P2) we have $E \in \mathcal{L}_{\mathcal{A}}(s)$. Consequently, if the simplified loop considers some class $D \in \mathcal{L}_{\mathcal{A}}(s)$, it will also consider each class E for which $D \rightsquigarrow_H E$.

Also note that, if a class C is unsatisfiable (lines 10–16), then our algorithm propagates the unsatisfiability of C to each class D that reaches C in K. This allows our algorithm to identify unsatisfiable classes without performing actual satisfiability tests, which significantly reduces the number of satisfiability tests needed to classify ontologies with many unsatisfiable classes; for example, as we discuss in Section 6 in more detail, this significantly improves the performance of classifying the FMA ontology [6]. Note, however, that such classes are removed from ToTest and are never considered again after line 7 since $\langle D, \text{owl:Nothing}\rangle \in K$. Hence, the algorithm might miss the opportunity to create a pre-model for a satisfiable parent class E of one of these classes; this, in turn, might render the algorithm incomplete since $P|_E$ might not contain all possible subsumers of E. This issue is solved by adding all parents E of an unsatisfiable class D to ToTest in lines 13–15.

Example 9. Let O contain axioms (14)–(17).

$$\text{SubClassOf}(C\ \text{ObjectSomeValuesFrom}(op\ D)) \quad (14)$$
$$\text{ObjectPropertyDomain}(op\ E) \quad (15)$$
$$\text{SubClassOf}(D\ \text{ObjectUnionOf}(E\ F)) \quad (16)$$
$$\text{SubClassOf}(G\ \text{ObjectSomeValuesFrom}(op2\ D)) \quad (17)$$

Assume that we want to classify the classes of the ontology using Algorithm 1. Since O is satisfiable, the algorithm proceeds

by calling Algorithm 2 with arguments O and $\mathbf{C}_O$.

Algorithm 2 determines that O contains no explicit subsumptions, so it initialises K to the empty relation. Thus, in the extracted hierarchy all classes different from owl:Thing are above owl:Nothing, so ToTest initially contains C, D, E, F, and G. Let us assume that the algorithm next checks the satisfiability of C by producing a pre-model $\mathcal{A}$ for C with root s_0 that satisfies property (P2). Due to axiom (14), s_0 must have an op-successor, say s_1, that is an instance of D. Since $D \in \mathcal{L}_{\mathcal{A}}(s_1)$, the pre-model $\mathcal{A}$ also witnesses the satisfiability of D. Due to axiom (16), $\mathcal{A}$ contains ClassAssertion(E s_1) or ClassAssertion(F s_1); let us assume that the former is the case. Using the information from P, our algorithm modifies P to ensure $P|_C = \{C\}$ and $P|_E = P|_D = \{D, E\}$. Let us assume that D is selected next; now $P|_D \neq \emptyset$, which means that a pre-model for D has already been encountered, so no test is performed for D. The algorithm then continues processing classes in ToTest, and at some point it selects G and constructs a pre-model $\mathcal{A}$ for G with root s_0 that satisfies property (P2). Due to axiom (17), s_0 has an $op2$ successor, say s_1, that is an instance of D; that is, $D \in \mathcal{L}_{\mathcal{A}}(s_1)$. Let us assume, however, that axiom (16) is satisfied in $\mathcal{A}$ by ClassAssertion(F s_1). Now, $P|_D \neq \emptyset$, so the classes in $\mathcal{L}_{\mathcal{A}}(s_1)$ are used to prune $P|_D$: since $E \notin \mathcal{L}_{\mathcal{A}}(s_1)$, relation P is modified to ensure $E \notin P|_D$; thus, $P|_D$ is left to contain only D. Consequently, $P|_D = \emptyset$ after the final cleanup—that is, all subsumers of D are known despite the fact that the satisfiability of D has never been tested explicitly by Algorithm 2 in line 8, thus illustrating the benefit of exploiting the information generated by satisfiability tests. $\diamond$

Next, we prove the correctness of Algorithm 2.

Lemma 10. *When applied to a satisfiable ontology O and a set of classes $S \subseteq \mathbf{C}_O$, Algorithm 2 terminates. Let K and P be the relations produced by the algorithm; then, for all classes $A, B \in S$, the following properties hold:*

1. *$A \rightsquigarrow_K B$ implies $O \models A \sqsubseteq B$.*
2. *If A is unsatisfiable, then $A \rightsquigarrow_K$ owl:Nothing.*
3. *If A is satisfiable and $O \models A \sqsubseteq B$, then either $A \rightsquigarrow_K B$ or a class A' exists such that $A \rightsquigarrow_K A'$, $\langle A', B\rangle \in P$, and $O \models A' \sqsubseteq B$.*

Proof. We show that the algorithm terminates after a finite number of steps. Note that function buildPreModel terminates for each set of assertions and each OWL 2 ontology [26]. Moreover, for each class C removed from ToTest, the algorithm adds $\langle C, \text{owl:Nothing}\rangle$ to K if C is unsatisfiable, or in line 21 it extends P with pairs of the form $\langle C, E\rangle$ which makes $P|_C$ not empty; note that in this case the algorithm also adds $\langle C, C\rangle$ to P. Furthermore, $\langle C, C\rangle$ is never removed from P in line 23, so $P|_C$ never becomes empty in future iterations of the while-loop. Thus, in the worst case, the algorithm considers each class in S once and then terminates.

(Claim 1) Pair $\langle D, \text{owl:Nothing}\rangle$ can be added to K in line 11, but then D is unsatisfiable. Alternatively, pair $\langle C, D\rangle$ can be added to K in line 18, but then $O \models C \sqsubseteq D$ by property (P1) from Section 2.3. Since these are the only places where pairs are added to K, Claim 1 clearly holds.

In order to prove Claims 2 and 3, we first prove two useful properties. Let H be as specified in line 2 and let A be an arbitrary class occurring in H; then, the following properties hold at the beginning of each iteration of the while-loop.

> ($\blacklozenge$): If A is satisfiable and $P|_A \neq \emptyset$, then, for each class $B \in S$ such that $O \models A \sqsubseteq B$, we have $B \in P|_A$.
>
> ($\clubsuit$): If $P|_A = \emptyset$ and $A \not\rightsquigarrow_K$ owl:Nothing, then there exists a class $F \in$ ToTest such that $F \not\rightsquigarrow_K$ owl:Nothing, $F \rightsquigarrow_H A$, and $P|_G = \emptyset$ for each class G such that $F \rightsquigarrow_H G$ and $G \rightsquigarrow_H A$.

For property ($\blacklozenge$), consider an arbitrary class A occurring in H. Set $P|_A$ can become nonempty in line 21; but then, $P|_A = \text{possibleSubsumers}(S, D, s, \mathcal{A}, K)$ after the change, so $P|_A$ clearly satisfies property ($\blacklozenge$) after the change. Alternatively, set $P|_A$ can be reduced in line 23 due to the removal of $\langle A, B\rangle$; but then, we have $O \not\models A \sqsubseteq B$, so set $P|_A$ clearly satisfies ($\blacklozenge$) after the change.

We next show by induction on the iterations of the while-loop that an arbitrary class A occurring in H satisfies property ($\clubsuit$).

Base Case: At the the beginning of the first iteration, ToTest contains all classes of H 'above' owl:Nothing; hence, for an arbitrary class $F \in$ ToTest, we have $F \not\rightsquigarrow_H$ owl:Nothing. Therefore, if $P|_A = \emptyset$ and $A \not\rightsquigarrow_H$ owl:Nothing, then property ($\clubsuit$) is satisfied for $F = A$.

Induction Step: Assume that property ($\clubsuit$) holds for A at the beginning of iteration i. We show that property ($\clubsuit$) also holds for A at the end of the iteration—that is, the property holds at the beginning of iteration $i + 1$. The claim is nontrivial only if $P|_A = \emptyset$ and $A \not\rightsquigarrow_K$ owl:Nothing. Since A satisfies the induction hypothesis, there exists a class $F \in$ ToTest such that $F \not\rightsquigarrow_K$ owl:Nothing, $F \rightsquigarrow_H A$, and $P|_G = \emptyset$ for each class G such that $F \rightsquigarrow_H G$ and $G \rightsquigarrow_H A$. Let C be an arbitrary class chosen in line 6. If C does not satisfy the condition in line 7, then $P|_C \neq \emptyset$ or $C \rightsquigarrow_K$ owl:Nothing, so $C \neq F$, and thus F satisfies property ($\clubsuit$) for A at the end of the iteration. If C satisfies the condition in line 7, we have two possibilities.

First, assume that C is satisfiable, and let $\mathcal{A}$ be the pre-model obtained in line 8. For an arbitrary class D, if $P|_D = \emptyset$ at the beginning, but not at the end of the loop, then by the condition in line 19 we have $P|_E \neq \emptyset$ at the end of the loop for each class E such that $D \rightsquigarrow_H E$. Consequently, if $P|_G \neq \emptyset$ at the end of the loop for some class G such that $F \rightsquigarrow_H G$ and $G \rightsquigarrow_H A$, then $P|_A \neq \emptyset$ at the end of the loop as well, so property ($\clubsuit$) is satisfied for A in line 26. Otherwise, since lines 17–26 never add a pair of the form $\langle F, \text{owl:Nothing}\rangle$ to K, class F satisfies property ($\clubsuit$) for A in line 26.

Second, assume that C is unsatisfiable; then, property ($\clubsuit$) can be affected only if $F \rightsquigarrow_H C$. To summarise, we have $F \rightsquigarrow_H C$ and $F \rightsquigarrow_H A$, where H is a directed acyclic relation; but then, a 'highest' class D in H exists that occurs on the path from F to C and on the path from F to A. More formally, there exists class D such that

- $F \rightsquigarrow_H D$,
- $D \rightsquigarrow_H A$,

- $D \rightsquigarrow_H C$, and
- for each class D' different from D such that $D \rightsquigarrow_H D'$ and $D' \rightsquigarrow_H A$, we have $D' \not\rightsquigarrow_H C$.

Furthermore, since we have $F \not\rightsquigarrow_H$ owl:Nothing, we also have $D \not\rightsquigarrow_K$ owl:Nothing. Class D will clearly eventually be considered in line 10. If $D = A$, then $\langle A, \text{owl:Nothing}\rangle$ is added to K in line 11, so A trivially satisfies property (♣) at the end of the iteration. If $D \neq A$, a class E exists such that $\langle D, E\rangle \in H$ and $E \rightsquigarrow_H A$; since $A \not\rightsquigarrow_K$ owl:Nothing, for each such E we have $E \not\rightsquigarrow_K$ owl:Nothing; furthermore, by property (♣), we have $P|_G = \emptyset$ for each class G such that $E \rightsquigarrow_H G$ and $G \rightsquigarrow_H A$. At least one such E is considered in lines 13–15 and is added to ToTest in line 14, so E satisfies property (♣) for A at the end of the iteration.

This completes the proof of property (♣) and we next prove Claims 2 and 3.

(Claim 2) Consider an arbitrary unsatisfiable class $A \in S$. By the definition of the hierarchy function, a class A' occurring in H exists such that $A \in \rho(A')$. If A occurs in K, then we clearly have $A \rightsquigarrow_K A'$ and $A' \rightsquigarrow_K A$. Assume that A does not occur in K. Then, since A is unsatisfiable (i.e., $A \neq$ owl:Thing) and owl:Thing and owl:Nothing are the only classes that can occur in H but not in K, we have $A =$ owl:Nothing; but then $A' = A$, and so we have $A \rightsquigarrow_K A'$ and $A' \rightsquigarrow_K A$ (remember that each class is reachable from itself). Class A' satisfies property (♣); furthermore, we have ToTest $= \emptyset$ upon termination, so by the contrapositive of property (♣) either $A' \rightsquigarrow_K$ owl:Nothing or $P|_{A'} \neq \emptyset$. Note, however, that unsatisfiable classes never appear in pre-models, so the algorithm never adds a pair of the form $\langle A', C\rangle$ to P. Thus, $P|_{A'} = \emptyset$, so we have $A' \rightsquigarrow_K$ owl:Nothing, and consequently $A \rightsquigarrow_K$ owl:Nothing as well.

(Claim 3) Consider an arbitrary satisfiable class $A \in S$ and an arbitrary class $B \in S$ such that $O \models A \sqsubseteq B$. By the definition of the hierarchy function, a class A' occurring in H exists such that $A \in \rho(A')$. If A occurs in K, then we clearly have $A \rightsquigarrow_K A'$ and $A' \rightsquigarrow_K A$. Assume that A does not occur in K; since A is satisfiable (i.e., $A \neq$ owl:Nothing) and owl:Thing and owl:Nothing are the only classes that can occur in H but not in K, we have $A =$ owl:Thing; but then $A' = A$, and so we have $A \rightsquigarrow_K A'$ and $A' \rightsquigarrow_K A$ (remember that each class is reachable from itself). Class A' satisfies property (♣); furthermore, we have ToTest $= \emptyset$ upon termination, so by the contrapositive of property (♣) we have that either $A' \rightsquigarrow_K$ owl:Nothing or $P|_{A'} \neq \emptyset$. By Claim 1 and the fact that A is satisfiable the former cannot be the case, so we have $P|_{A'} \neq \emptyset$. But then, by property (♦) we have $B \in P|_{A'}$ after line 28, so Claim 3 holds at this point. Pair $\langle A', B\rangle$ can be removed from P in line 29, but then $A' \rightsquigarrow_K B$, and so we have $A \rightsquigarrow_K B$; thus, Claim 3 holds after line 29 as well. □

3.3. The Classification Phase

In the classification phase, our algorithm determines which of the possible subsumptions in P actually hold. This is done as shown in Algorithm 3. Each class C for which there are possible subsumptions is processed iteratively (lines 1–34). Since K is initialised with explicit subsumptions in O, it is often the case that no class in $P|_C$ is a subsumer of C, so identifying such situations quickly can significantly reduce the total number of subsumption tests. This is done by trying to construct a pre-model that satisfies C and no $D_i \in P|_C$ (lines 2 and 3); if a pre-model $\mathcal{A}$ can be constructed, then no D_i is a subsumer of C, so all pairs $\langle C, D_i\rangle$ are removed from P (line 5) and P is further pruned using the information from $\mathcal{A}$ (line 6). If at least one $D_i \in P|_C$ is a subsumer of C, the algorithm first reads the known subsumers of C off the returned set of assertions $\mathcal{A}$ (line 8), and it prunes P by removing the known subsumers of C (line 9). As explained earlier, treating assertion ClassAssertion(F s_0) as being derived nondeterministically allows us to identify the known subsumers of C during a subsumption test. If C still has possible subsumers, then C is inserted into the hierarchy constructed thus far (lines 11–30). In order to reduce the number of subsumption tests, the classes in $P|_C$ are arranged into a hierarchy H_C that is compatible with K (line 11); then, H_C is traversed using a variant of the ET algorithm (lines 12–30). To this end, a queue Q is initialised to contain all children of owl:Thing in H_C (line 12); this prevents the algorithm from checking the trivial subsumption between C and owl:Thing. As long as Q is not empty (lines 13–30), the head D of Q is popped off Q (line 14) with the intention to check whether D subsumes C. The algorithm does not process the class D if this subsumption was discovered to be known (line 15) or if D was removed from $P|_C$ since H_C was constructed; this can happen if D is added to Q more than once due to the presence of several paths from D to owl:Thing in H_C or if D was discovered to be a subsumer of C in a previously constructed pre-model. Otherwise, the subsumption between C and D is tested by trying to construct a pre-model satisfying C but not D (line 19). If such a pre-model $\mathcal{A}$ can be constructed, then $\langle C, D\rangle$ together with all known subclasses of D are removed from P (line 21), and P is further pruned using the information from $\mathcal{A}$ (line 22). In contrast, if the subsumption holds, this is recorded in K (line 24), and each child E of D in H_C is added to Q (line 25) in order to continue the traversal of H_C. In either case, known subsumers of C are read off $\mathcal{A}$ and P is pruned accordingly (lines 27 and 28). Finally, P is pruned as discussed in Section 3.1 for all newly discovered known subsumptions (line 32).

Please note that, if the pre-models constructed in lines 3 and 19 satisfy property (P2) from Section 2.3, then lines 5 and 21 are subsumed by lines 6 and 22, respectively. In particular, for each class D_i removed in line 5, by property (P2) we have $D_i \notin \mathcal{L}_{\mathcal{A}}(s_0)$, so $\langle C, D_i\rangle$ is removed from P in line 6. Similarly, for each class E removed in line 21, by property (P2) we have $E \notin \mathcal{L}_{\mathcal{A}}(s_0)$, so $\langle C, E\rangle$ is removed from P in line 22.

In contrast to the ET algorithm, our algorithm does not include a bottom-up phase. This considerably simplifies the implementation, as one does not need data structures that allow retrieval of the predecessors of a class C in K and P: the algorithm can be efficiently implemented by explicitly keeping track only of successor links. Furthermore, unlike in the bottom-up phase of the ET algorithm, our algorithm never iterates over the direct superclasses of owl:Nothing, which significantly reduces the cost of data structure traversal.

Algorithm 3 processRemainingClasses(K, P, O, S)

Input: binary relations K and P, an ontology O and a set of classes S

```
1: while some C ∈ S exists such that P|_C ≠ ∅ do
2:   F := ObjectComplementOf(ObjectUnionOf(D_1 ... D_n)) where {D_1, ..., D_n} = P|_C
3:   A := buildPreModel({ClassAssertion(C s_0)}, {ClassAssertion(F s_0)}, O)
4:   if unsatisfiable ∉ A then                //no D_i ∈ P|_C subsumes C
5:     remove each ⟨C, D_i⟩ from P such that D_i ∈ P|_C
6:     P := prune(P, O, A, K)
7:   else
8:     K := K ∪ {⟨C, D⟩ | D ∈ knownSubsumers(S, s_0, A)}
9:     remove each ⟨C, E⟩ from P such that C ⇝_K E
10:    if P|_C ≠ ∅ then
11:      (V_C, H_C, ρ_C) := hierarchy(P|_C, K, owl:Nothing, owl:Thing)
12:      initialise a queue Q to contain all D with ⟨D, owl:Thing⟩ ∈ H_C
13:      while Q ≠ ∅ do
14:        remove the head D from Q
15:        if C ⇝_K D then
16:          add to the end of Q each E such that ⟨E, D⟩ ∈ H_C
17:        else if D ∈ P|_C then
18:          F := ObjectComplementOf(D)
19:          A := buildPreModel({ClassAssertion(C s_0)}, {ClassAssertion(F s_0)}, O)
20:          if unsatisfiable ∉ A then   // O ⊭ C ⊑ D
21:            remove each ⟨C, E⟩ from P such that E ⇝_K D
22:            P := prune(P, O, A, K)
23:          else
24:            add ⟨C, D⟩ to K
25:            add to the end of Q each E such that ⟨E, D⟩ ∈ H_C
26:          end if
27:          add ⟨C, D⟩ to K for each D ∈ knownSubsumers(S, s_0, A)
28:          remove each ⟨C, E⟩ from P such that C ⇝_K E
29:        end if
30:      end while
31:    end if
32:    remove each ⟨E_1, E_2⟩ from P such that E_1 ⇝_K E_2
33:  end if
34: end while
```

Lemma 11. *Let O be a satisfiable ontology, and let K and P be the relations obtained by applying Algorithm 2 to O and a set of classes $S \subseteq \mathbf{C}_O$. Then, applying Algorithm 3 to K, P, O, and S terminates. Let K be the relation produced by the algorithm; then, for all classes $A, B \in S$, the following properties hold:*

1. *$A \rightsquigarrow_K B$ implies $O \models A \sqsubseteq B$.*
2. *If A is unsatisfiable, then $A \rightsquigarrow_K$ owl:Nothing.*
3. *If A is satisfiable and $O \models A \sqsubseteq B$, then $A \rightsquigarrow_K B$.*

Proof. First, we prove that the algorithm terminates. Consider an arbitrary class C selected in line 1. We show that $P|_C = \emptyset$ at the end of the outer while-loop; since the algorithm never adds pairs to P, this clearly implies termination. Let $\mathcal{A}$ be the pre-model obtained in line 3. If $\mathcal{A}$ satisfies the condition in line 4, then each pair $\langle C, D_i \rangle$ is removed from P in line 5 thus making $P|_C$ empty. We next assume that the condition in line 4 is not satisfied, so the algorithm proceeds with lines 8–30. If $P|_C \neq \emptyset$ after line 9, the algorithm proceeds with lines 11–30. In particular, the algorithm constructs a hierarchy H_C containing each $D \in P|_C$, and then it traverses H_C using breadth-first search. Since H_C is acyclic, the while loop in lines 13–30 terminates: each class D occurring in H_C can be added to Q only as many times as there are paths from owl:Thing to D in H_C. We next show that $P|_C = \emptyset$ in line 34 of the outer while loop. Since tuples are never added to P, this clearly implies that the algorithm terminates.

To show that $P|_C = \emptyset$ in line 34, we first prove that the following three properties hold at the end of each iteration of the inner while loop—that is, after line 29.

($\sharp$): For each class E such that $C \rightsquigarrow_K E$, we have $E \notin P|_C$.

($\star$): For each class D selected in line 14, we have $D \notin P|_C$.

($\spadesuit$): For each class $F \in P|_C$, a class $G \in Q$ exists with $F \rightsquigarrow_K G$.

For property ($\sharp$), note that the property holds before the while loop (i.e., after line 12) due to pruning in line 9; furthermore, K can be extended in an iteration only in line 24, but then line 28 ensures that property ($\sharp$) holds at the end of the iteration.

For property ($\star$), let D be an arbitrary class selected in line 14. If the condition in line 15 holds, then $D \notin P|_C$ due to prop-

erty ($\sharp$). If the condition in line 15 does not hold, then either $\langle C, D\rangle$ is removed from P in line 21, or $\langle C, D\rangle$ is added to K in line 24 and so $\langle C, D\rangle$ is removed from P in line 28; either way, $D \notin P|_C$ at the end of the iteration.

For property ($\spadesuit$), note that the property clearly holds after Q is initialised in line 12. Consider now an arbitrary class F such that $F \in P|_C$ at the beginning of the iteration, and let G be the class that satisfies property ($\spadesuit$) for F. Furthermore, let D be the class selected in line 14. If $D \neq G$, then property ($\spadesuit$) clearly holds for F at the end of the iteration. If $D = G$, then we have the following possibilities: some E such that $F \rightsquigarrow_K E$ is added to Q in line 16 or line 25, or $\langle C, F\rangle$ is removed from P in line 21. In all cases, property ($\spadesuit$) holds for F at the end of the iteration.

We now complete the proof that $P|_C = \emptyset$ in line 34. In particular, since $Q = \emptyset$ after line 30, no $F \in P|_C$ can exist without violating property ($\spadesuit$). Thus, the algorithm terminates, and we have $P = \emptyset$ upon termination.

To complete the proof of this lemma, we next show that relations K and P satisfy at all times during the algorithm's execution the following properties for all classes $A, B \in S$.

1. $A \rightsquigarrow_K B$ implies $O \models A \sqsubseteq B$.
2. If A is unsatisfiable, then $A \rightsquigarrow_K$ owl:Nothing.
3. If A is satisfiable and $O \models A \sqsubseteq B$, then either $A \rightsquigarrow_K B$ or a class A' exists such that $A \rightsquigarrow_K A'$, $\langle A', B\rangle \in P$, and $O \models A' \sqsubseteq B$.

Initially, these properties are satisfied as a consequence of Lemma 10.

(Property 1) A pair $\langle A, B\rangle$ can be added to K in lines 8 and 27, but then $O \models A \sqsubseteq B$ holds since $\mathcal{A}$ satisfies property (P1) from Section 2.3. Alternatively, a pair $\langle A, B\rangle$ can be added to K in line 24, but then $O \models A \sqsubseteq B$ holds as a consequence of the subsumption check in line 19. Thus, property 1 holds at any point during the algorithm's execution.

(Property 2) Pairs are never removed from K, so Property 2 never ceases to hold for an arbitrary unsatisfiable class $A \in S$.

(Property 3) Consider arbitrary classes $A, B \in S$ such that A is satisfiable and $O \models A \sqsubseteq B$. If $A \rightsquigarrow_K B$ holds at some point, then $A \rightsquigarrow_K B$ never ceases to hold because pairs are never removed from K. Furthermore, assume that a class A' exists such that $A \rightsquigarrow_K A'$, $\langle A', B\rangle \in P$, and $O \models A' \sqsubseteq B$. Property 3 might cease to hold for A and B only after a modification of P, so we next consider all possible ways in which that could happen.

- If $\langle A', B\rangle$ is removed from P in line 9, 28, or 32, then we have $A' \rightsquigarrow_K B$; thus, we have $A \rightsquigarrow_K B$, so Property 3 holds after the removal.

- If $\langle C, D_i\rangle$ is removed from P in line 5, then $O \not\models C \sqsubseteq D_i$; thus, we have $\langle C, D_i\rangle \neq \langle A', B\rangle$, so Property 3 holds after the removal.

- If $\langle C, E\rangle$ is removed from P in line 21, by $O \not\models C \sqsubseteq D$ and $E \rightsquigarrow_K D$ we have $O \not\models C \sqsubseteq E$; thus, $\langle C, E\rangle \neq \langle A', B\rangle$, so Property 3 holds after the removal.

- If $\langle F_1, F_2\rangle$ is removed from P in lines 6 and 22, by the definition of prune we have $O \not\models F_1 \sqsubseteq F_2$. Thus, we have $\langle F_1, F_2\rangle \neq \langle A', B\rangle$, so Property 3 holds after the removal.

Upon termination we have $P = \emptyset$, which together with Properties 1–3 clearly implies the claim of this lemma. □

Theorem 12. *For each ontology O, Algorithm 1 terminates and it correctly computes the class hierarchy of O.*

Proof. The claim holds trivially if O is unsatisfiable, so let us assume that O is satisfiable. Termination is an immediate consequence of the fact that Algorithms 2 and 3 terminate on O. Finally, correctness also follows straightforwardly from Lemma 11 and the fact that Algorithm 1 produces a hierarchy of $\mathbf{C}_O$ w.r.t. K, owl:Nothing, and owl:Thing. □

4. Object Property Classification

To the best of our knowledge, classification of object properties has not been discussed in the literature, and all ontology reasoners we are aware of construct the object property hierarchy simply by computing the reflexive–transitive closure of the asserted object property hierarchy. Such an algorithm requires no complex reasoning and it can be easily implemented; however, it is incomplete even for very simple sublanguages of OWL. We demonstrate this by means of an example that uses existential restrictions (ObjectSomeValuesFrom), functional properties, and property hierarchies. In the rest of this section, we use $op_{(i)}$ to denote an object property and $ope_{(i)}$ to denote an object property expression.

Example 13. Consider axioms (18)–(21).

$$\begin{array}{l}\text{SubClassOf(}\\ \quad\text{ObjectSomeValuesFrom(}op_1\text{ owl:Thing)}\\ \quad\text{ObjectSomeValuesFrom(}op_2\text{ owl:Thing)}\\ \text{)}\end{array} \tag{18}$$

$$\text{SubObjectPropertyOf(}op_1\ op_3\text{)} \tag{19}$$

$$\text{SubObjectPropertyOf(}op_2\ op_3\text{)} \tag{20}$$

$$\text{FunctionalObjectProperty(}op_3\text{)} \tag{21}$$

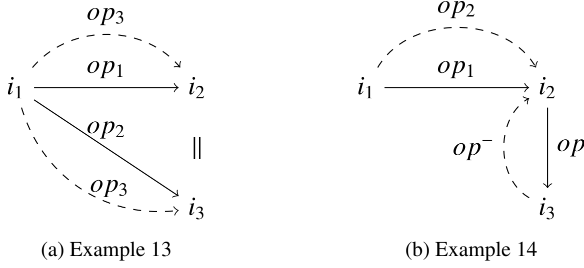
These axioms entail $op_1 \sqsubseteq op_2$: if i_2 is an op_1-successor of i_1 in an interpretation I, then axiom (18) requires the existence of an op_2-successor i_3 of i_1 in I; since both op_1 and op_2 are subproperties of op_3 and op_3 is functional, then i_3 is equal to i_2, so i_2 is also an op_2-successor of i_1. This is shown graphically in the left part of Figure 3. ◇

Furthermore, the following example demonstrates that subsumption relationships between object properties can also be derived due to an interaction between object property chains and inverse properties (ObjectInverseOf).

Example 14. Consider axioms (22)–(23).

$$\begin{array}{l}\text{SubClassOf(owl:Thing}\\ \quad\text{ObjectSomeValuesFrom(}op\text{ owl:Thing)}\\ \text{)}\end{array} \tag{22}$$

$$\begin{array}{l}\text{SubObjectPropertyOf(}\\ \quad\text{ObjectPropertyChain(}op_1\ op\text{ ObjectInverseOf(}op\text{))}\\ \quad op_2\\ \text{)}\end{array} \tag{23}$$



(a) Example 13 (b) Example 14

Figure 3: Graphical illustration of the models created in Examples 13 and 14. Dashed arrows indicate inferred relations, and op^{-} is the inverse of the object property op.

If i_1 has an op_1-successor i_2 in a model I, axiom (22) ensures that i_2 has an op-successor i_3; hence, $\langle i_1, i_2 \rangle$ is in the interpretation of op_1, $\langle i_2, i_3 \rangle$ is in the interpretation of op, and $\langle i_3, i_2 \rangle$ is in the interpretation of ObjectInverseOf(op). By axiom (23), then $\langle i_1, i_2 \rangle$ is in the interpretation of op_2 so, consequently, the ontology entails $op_1 \sqsubseteq op_2$. This is shown graphically in the right part of Figure 3. ◇

One might assume that object properties can be classified correctly and efficiently by modifying the classification algorithm from Section 3 in the obvious way: to check whether an object property op_1 subsumes an object property op_2, one should construct a model satisfying assertions (24) and (25) where a and b are fresh individuals; furthermore, to update relations P and K, one should consider labels of individual pairs instead of single individuals.

$$\text{ObjectPropertyAssertion}(op_1\ a\ b) \tag{24}$$

$$\text{NegativeObjectPropertyAssertion}(op_2\ a\ b) \tag{25}$$

Somewhat surprisingly, such an algorithm is incomplete due to a problem with *complex properties*—that is, properties that are transitive or are defined using a complex property inclusion axiom. In all (hyper)tableau calculi known to us, axioms defining complex properties are not handled directly, but via an equi-satisfiable encoding [28, 27, 29]. For example, consider an ontology that contains the following axioms:

$$\text{TransitiveObjectProperty}(op) \tag{26}$$

$$\text{ObjectPropertyAssertion}(op\ a\ b) \tag{27}$$

$$\text{ObjectPropertyAssertion}(op\ b\ c) \tag{28}$$

$$\text{NegativeObjectPropertyAssertion}(op\ a\ c) \tag{29}$$

This ontology is clearly unsatisfiable. To determine this, one might expect a (hyper)tableau algorithm to derive

$$\text{ObjectPropertyAssertion}(op\ a\ c) \tag{30}$$

from (26)–(28), and then to derive a contradiction from (29) and (30). To the best of our knowledge, however, no (hyper)tableau calculus works in such a way. The addition of transitively implied object property assertions such as (30) is not compatible with *blocking* [26]—a technique used to ensure termination of pre-model construction. Instead, all calculi known to us use an encoding that simulates the effects of axioms such as (26). In particular, each negative object property assertion such as (29) is replaced with an equivalent axiom (31).

$$\begin{array}{l}\text{ClassAssertion(}\\ \quad \text{ObjectAllValuesFrom(}op\\ \quad\quad \text{ObjectComplementOf(ObjectOneOf(}c\text{))}\\ \quad)\\ \quad a\\ )\end{array} \tag{31}$$

Next, all axioms containing ObjectAllValuesFrom classes are transformed in a certain way; for example, axiom (31) is replaced with the following axioms, where Q is a fresh class:

$$\text{ClassAssertion(ObjectAllValuesFrom(}op\ Q\text{)}\ a) \tag{32}$$

$$\text{SubClassOf(}Q\ \text{ObjectComplementOf(ObjectOneOf(}c\text{)))} \tag{33}$$

$$\text{SubClassOf(}Q\ \text{ObjectAllValuesFrom(}op\ Q\text{))} \tag{34}$$

Intuitively, axioms (32)–(34) ensure that each individual in a pre-model reachable via op from a is an instance of

$$\text{ObjectComplementOf(ObjectOneOf(}c\text{))},$$

which captures the effect of axioms (26) and (31). Thus, axioms (26), (27), and (32)–(34) imply a contradiction, as required; however, note that no axiom forces a (hyper)tableau calculus to derive (30). Thus, a pre-model is not guaranteed to contain all implied object property assertions for complex properties, which adversely affects the completeness of our classification algorithm from Section 3: due to missing property assertions, the set of possible subsumers P might not be correctly initialised, or certain subsumptions might be incorrectly pruned from P. To summarise, the modified classification algorithm will correctly classify object properties that are not complex, but it might fail to discover certain subsumptions involving at least one complex object property.

In order to overcome these issues, we developed a new property classification technique that reduces object property classification to standard class classification. Any complete class classification algorithm (such as the one described in Section 3) can be used to classify the resulting ontology.

Definition 15. *Let O be an OWL 2 ontology, let C_f be a fresh class not occurring in O, let a be a fresh individual not occurring in O, and let τ be an injective function that maps each object property expression $ope \in \mathbf{OPE}_O$ into a class $\tau(ope)$ as follows:*

- *τ(owl:topObjectProperty) = owl:Thing,*
- *τ(owl:bottomObjectProperty) = owl:Nothing, and*
- *$\tau(ope)$ is a fresh distinct class not occurring in $\mathbf{C}_O \cup \{C_f\}$ for each $ope \in \mathbf{OPE}'_O$.*

Then, O_τ is the ontology obtained by extending O with assertion (35) *and an instance of axiom* (36) *for each $ope \in \mathbf{OPE}'_O$.*

$$\textit{ClassAssertion}(C_f\ a) \tag{35}$$

$$\textit{EquivalentClasses}(\tau(ope) \quad \textit{ObjectSomeValuesFrom}(ope\ C_f)) \tag{36}$$

As we show in Theorem 16, the encoding from Definition 15 allows us to check $O \models ope_1 \sqsubseteq ope_2$ by equivalently checking $O_\tau \models \tau(ope_1) \sqsubseteq \tau(ope_2)$.

Thus, for O, op_1 and op_2 defined as in Example 13, we can check whether $O \models op_1 \sqsubseteq op_2$ holds by checking whether $O_\tau \models \tau(op_1) \sqsubseteq \tau(op_2)$ holds, where O_τ is obtained by extending O with axioms (37)–(38).

$$\text{EquivalentClasses}(\tau(op_1) \quad \text{ObjectSomeValuesFrom}(op_1\ C_f)) \tag{37}$$

$$\text{EquivalentClasses}(\tau(op_2) \quad \text{ObjectSomeValuesFrom}(op_2\ C_f)) \tag{38}$$

The latter can be checked as usual, by trying to construct a pre-model for assertions (39)–(40), where s_0 is a fresh individual.

$$\text{ClassAssertion}(\tau(op_1)\ s_0) \tag{39}$$

$$\text{ClassAssertion}(\text{ObjectComplementOf}(\tau(op_2))\ s_0) \tag{40}$$

Since O_τ contains axiom (37), assertion (39) implies that s_0 must have an op_1-successor s_1 that is an instance of C_f; now, if the axioms in O imply that s_1 is necessarily an op_2-successor of s_0 as well, then axiom (38) from O_τ implies that s_0 is an instance of $\tau(op_2)$, which contradicts assertion (40).

Note that the axioms for complex properties in O_τ are subject to the encoding of complex properties described above. The additional axioms in O_τ might not look as if they contain an ObjectAllValuesFrom class, but this becomes obvious if the axioms are normalised. Any (hyper)tableau reasoner will preprocess these axioms before applying the actual reasoning calculus. For example, axiom (37) is split into the following two axioms:

$$\text{SubClassOf}(\tau(op_1)\ \text{ObjectSomeValuesFrom}(op_1\ C_f)) \tag{41}$$

$$\text{SubClassOf}(\text{ObjectSomeValuesFrom}(op_1\ C_f)\ \tau(op_1)) \tag{42}$$

These are subsequently reformulated as follows:

```
SubClassOf(owl:Thing
  ObjectUnionOf(
    ObjectComplementOf(τ(op1))
    ObjectSomeValuesFrom(op1 Cf)
  )
)                                              (43)
```

```
SubClassOf(owl:Thing
  ObjectUnionOf(
    ObjectComplementOf(
      ObjectSomeValuesFrom(op1 Cf)
    )
    τ(op1)
  )
)                                              (44)
```

The latter axiom in finally brought into negation-normal form as follows:

```
SubClassOf(owl:Thing
  ObjectUnionOf(
    ObjectAllValuesFrom(op1
      ObjectComplementOf(Cf)
    )
    τ(op1)
  )
)                                              (45)
```

These transformations make it clear that axiom (37) contains an ObjectAllValuesFrom class expression, which triggers the encoding if op_1 is a complex property.

Furthermore, note that, by Definition 15, O_τ does not contain axioms of the form (36) for owl:topObjectProperty and owl:bottomObjectProperty. From a theoretical point of view, one could map these two properties via τ to fresh classes and then include the corresponding axioms of the form (36) in O_τ. The drawback of such an approach, however, is that O_τ then contains owl:topObjectProperty and owl:bottomObjectProperty even if O does not, and reasoning with these two properties can be difficult. By mapping owl:topObjectProperty and owl:bottomObjectProperty to owl:Thing and owl:Nothing, respectively, and not including in O_τ the corresponding axioms of the form (36), we ensure that reasoning with O_τ is usually not more difficult than reasoning with O.

Theorem 16. *Let O, τ, and O_τ be as in Definition 15, and let ope_1 and ope_2 be arbitrary object property expressions in $\mathbf{OPE}_O$. Then, $O \models ope_1 \sqsubseteq ope_2$ iff $O_\tau \models \tau(ope_1) \sqsubseteq \tau(ope_2)$.*

Proof. ($\Leftarrow$) We prove the contrapositive: if $O \not\models ope_1 \sqsubseteq ope_2$, then $O_\tau \not\models \tau(ope_1) \sqsubseteq \tau(ope_2)$. Assume that $O \not\models ope_1 \sqsubseteq ope_2$; then there exists a model I of O such that $I \not\models ope_1 \sqsubseteq ope_2$. Clearly, we have

$$ope_1 \neq \text{owl:bottomObjectProperty} \quad \text{and} \quad ope_2 \neq \text{owl:topObjectProperty}.$$

Let $\langle i_1, i_2 \rangle$ be a tuple of objects that is contained in the interpretation of ope_1 in I, but not in the interpretation of ope_2 in I. We conservatively extend I to I' by interpreting the symbols in O_τ that do not occur in O as follows:

- the interpretation of individual a in I' is i_2,
- the interpretation of C_f in I' contains only i_2, and
- for each $ope \in \mathbf{OPE}'_O$, the interpretation of $\tau(ope)$ in I' contains each i such that $\langle i, i_2 \rangle$ is contained in the interpretation of ope in I.

Interpretation I' clearly satisfies O, assertion (35), and all axioms of the form (36) in O_τ; thus, I' is a model of O_τ. If ope_1 = owl:topObjectProperty, then $\tau(ope_1)$ = owl:Thing, so i_1 is clearly in the interpretation of $\tau(ope_1)$; otherwise, since $\langle i_1, i_2 \rangle$ is in the interpretation of ope_1 and i_2 is in the interpretation of C_f, then i_1 is in the interpretation of $\tau(ope_1)$ by the construction of I'. Similarly, if ope_2 = owl:bottomObjectProperty,

then $\tau(ope_2)$ = owl:Nothing, so i_2 is clearly not in the interpretation of $\tau(ope_2)$; otherwise, since $\langle i_1, i_2 \rangle$ is not in the interpretation of ope_2, then i_1 is not in the interpretation of $\tau(ope_2)$ by the construction of I'. Consequently, $I' \not\models \tau(ope_1) \sqsubseteq \tau(ope_2)$, so $O_\tau \not\models \tau(ope_1) \sqsubseteq \tau(ope_2)$, as required.

($\Rightarrow$) We prove the contrapositive: if $O_\tau \not\models \tau(ope_1) \sqsubseteq \tau(ope_2)$, then $O \not\models ope_1 \sqsubseteq ope_2$. Assume that $O_\tau \not\models \tau(ope_1) \sqsubseteq \tau(ope_2)$; then a model I of O_τ exists where some i_1 is an instance of $\tau(ope_1)$ but not of $\tau(ope_2)$; furthermore, O_τ contains all axioms of O, so I is a model of O; finally, we clearly have

$$ope_1 \neq \text{owl:bottomObjectProperty} \quad \text{and}$$
$$ope_2 \neq \text{owl:topObjectProperty}.$$

Now, if ope_1 = owl:topObjectProperty, due to axiom (35), i_2 exists that is an instance i_2 of C_f in I, and, due to the semantics of owl:topObjectProperty, i_2 is an ope_1-successor of i_1 in I; otherwise, due to axiom (36) for ope_1, i_2 exists that is an ope_1-successor of i_1 and that is an instance of C_f in I. But then, subsumption $O \not\models ope_1 \sqsubseteq ope_2$ holds trivially for ope_2 = owl:bottomObjectProperty. Assume now that $ope_2 \neq$ owl:bottomObjectProperty and that i_2 is an ope_2-successor of i_1 in I. Axiom (36) for ope_2 implies that i_1 is an instance of $\tau(ope_2)$, so i_1 is an instance of $\tau(ope_2)$ in I, which is a contradiction. Consequently, i_2 is not an ope_2-successor of i_1—that is, $I \not\models ope_1 \sqsubseteq ope_2$—so $O \not\models ope_1 \sqsubseteq ope_2$, as required. □

Our procedure for classifying object properties is shown in Algorithm 4. As in the case of classification of classes, the algorithm first checks whether the given ontology is satisfiable. If not, a trivial hierarchy in which all object property expressions in O are subsumed by owl:bottomObjectProperty is returned; otherwise, the algorithm proceeds with the classification. The algorithm constructs a mapping τ from object properties to classes as in Definition 15 (line 5). Algorithm 4 next calls the procedure initialiseOPRelations, which is defined analogously to procedure initialiseRelations (Algorithm 2), but with the following differences:

1. Instead of calling explicitSubsumptions (line 1 in Algorithm 2), initialiseOPRelations extracts from O the explicit object property subsumptions and then initialises K with a tuple $\langle \tau(ope_1), \tau(ope_2) \rangle$ for all object property expressions ope_1 and ope_2 such that ope_1 is explicitly subsumed by ope_2.
2. All remaining steps in initialiseOPRelations are as in Algorithm 2, but O_τ is used instead of O.

Once K has been computed by processRemainingClasses, Algorithm 4 uses τ to map the classes in K back to a relation K' over object property expressions (line 9). Finally, the algorithm constructs the object property hierarchy based on the subsumptions between object property expressions in K' (line 10).

5. Data Property Classification

By a straightforward modification of Example 13, we can show that data properties cannot be classified by computing the reflexive–transitive closure of the explicitly stated data property inclusions; essentially, we just need to replace owl:Thing with rdfs:Literal. Thus, reasoning is needed in order to correctly classify data properties.

Interestingly, data property subsumption cannot be easily reduced to satisfiability. To test $O \models dp_1 \sqsubseteq dp_2$ with dp_1 and dp_2 data properties, we would need to construct a pre-model satisfying assertions

$$\text{DataPropertyAssertion}(dp_1\ i\ n) \quad (46)$$
$$\text{NegativeDataPropertyAssertion}(dp_2\ i\ n) \quad (47)$$

for i a fresh individual and n a literal representing an arbitrary element of the data domain. In OWL 2, however, there is no such thing as a literal with an arbitrary data value: all literals are given a fixed interpretation as specified by the OWL 2 datatype map. Note that we cannot select n as some fixed literal not occurring in the ontology; for example, if we selected n to be an integer not occurring in the ontology, we might get a contradiction if the ontology axioms state that the range of dp_1 is xsd:string.

We can solve this problem by introducing a special datatype D that is interpreted as an arbitrary subset of rdfs:Literal. More precisely, we define an ontology O containing D to be satisfiable if and only if D can be assigned an interpretation such that all axioms of O are satisfied. Then, we can reduce the problem of checking $O \models dp_1 \sqsubseteq dp_2$ to the problem of checking whether O extended with the following assertions is satisfiable, for i a fresh individual:

$$\text{ClassAssertion}(\text{DataSomeValuesFrom}(dp_1\ \mathsf{D})\ i) \quad (48)$$
$$\text{ClassAssertion}(\text{DataAllValuesFrom}(dp_2\ \text{DataComplementOf}(\mathsf{D}))\ i) \quad (49)$$

Datatype reasoning is commonly implemented using a procedure such as the one by Motik and Horrocks [30]. This procedure represents datatype constraints using assertions of the form $dt(s)$, $\neg dt(s)$ and $s_1 \not\approx s_2$, where dt is a datatype, and s, s_1 and s_2 are *concrete nodes*—placeholders for data values. Given a set of assertions $\mathcal{A}$, the procedure checks whether the concrete nodes occurring in $\mathcal{A}$ can be assigned data values that respect all constraints. Roughly speaking, for every set of concrete nodes $s_1, \ldots, s_n$ such that $\mathcal{A}$ contains $s_i \not\approx s_{i+1}$ for each $1 \leq i < n$, the procedure tries to identify distinct data values $v_1, \ldots, v_n$ such that the value v_i is contained in the interpretation of each datatype dt_j that occurs in $\mathcal{A}$ in an assertion $dt_j(s_i)$, and is not contained in the interpretation of any datatype dt_k that occurs in $\mathcal{A}$ in an assertion $\neg dt_k(s_i)$. This procedure can be extended to handle the datatype D as follows:

1. If $\mathcal{A}$ contains assertions $\mathsf{D}(s_1)$ and $\neg\mathsf{D}(s_2)$, but not the assertion $s_1 \not\approx s_2$, then $\mathcal{A}$ is extended with $s_1 \not\approx s_2$.
2. Assertions of the form $\mathsf{D}(s_1)$ and $\neg\mathsf{D}(s_2)$ are ignored when trying to assign data values to concrete nodes.

The first item ensures that concrete nodes s_1 and s_2 are not accidentally assigned the same value, and the second item ensures that D places no additional constraints on the values assigned to concrete nodes in $\mathcal{A}$.

Algorithm 4 ClassifyObjectProperties(O)

Input: an ontology O whose set of object property expressions $\mathbf{OPE}_O$ should be classified

1: $\mathcal{A}$:= buildPreModel($\emptyset, \emptyset, O$)
2: **if** unsatisfiable $\in \mathcal{A}$ **then**
3: **return** the trivial hierarchy in which each object property expression $ope \in \mathbf{OPE}_O$ is subsumed by owl:bottomObjectProperty
4: **end if**
5: construct a mapping τ and the ontology O_τ as in Definition 15
6: initialise S to the range of τ
7: (K, P) := initialiseOPRelations(O, S, τ)
8: processRemainingClasses(K, P, O_τ, S)
9: K' := mapClassesToProperties(K, τ)
10: **return** hierarchy($\mathbf{OPE}_O, K'$, owl:bottomObjectProperty, owl:topObjectProperty)

Even if a concrete node s is assigned a value from D, the procedure from [30] does not necessarily insert an assertion D(s) into each pre-model. We therefore cannot read non-subsumptions off pre-models, which prevents us from directly applying the classification algorithm from Section 3. We can, however, reduce data property classification to class classification similarly as in Section 4.

Note that, in OWL 2 DL ontologies, owl:topDataProperty can occur only in axioms SubDataPropertyOf, in which they can only play the role of a super-property [3, Section 11.2]. This ensures that owl:topDataProperty occurs only in tautologies, so an axiom of the form

EquivalentClasses(Q DataSomeValuesFrom(
owl:topDataProperty D)) (50)

where Q is the class to which owl:topDataProperty is mapped is not allowed in OWL 2 DL. Therefore, unlike in the case of object properties, we have no choice but to ensure that owl:topDataProperty is mapped to owl:Thing. The OWL 2 DL specification restricts the usage of owl:topDataProperty in order to ensure that consequences of an OWL 2 DL ontology do not depend on the choice of a datatype map, as long as the datatype map chosen contains all the datatypes occurring in the ontology [4, Theorem DS1]. Due to this restriction, however, no data property different from owl:topDataProperty can subsume owl:topDataProperty, unless the ontology is unsatisfiable. This, in turn, ensures that our encoding does not need to include an axiom analogous to (35): we need not ensure that the interpretation of D is not empty, which simplifies reasoning with D.

Definition 17. *Let O be an OWL 2 ontology, let* D *be the special datatype as discussed above, and let σ be an injective function that maps each data property $dp \in \mathbf{DP}_O$ into a class $\sigma(dp)$ as follows:*

- *σ(owl:topDataProperty) = owl:Thing,*
- *σ(owl:bottomDataProperty) = owl:Nothing, and*
- *$\sigma(dp)$ is a fresh distinct class not occurring in $\mathbf{C}_O$ for each $dp \in \mathbf{DP}'_O$.*

Then, O_σ is the ontology obtained by extending O with an instance of axiom (51) *for each $dp \in \mathbf{DP}'_O$.*

EquivalentClasses($\sigma(dp)$ DataSomeValuesFrom(dp D)) (51)

The following theorem shows that the reduction is indeed correct.

Theorem 18. *Let O, σ, and O_σ be as in Definition 17, and let dp_1 and dp_2 be arbitrary data properties in $\mathbf{DP}_O$. Then, $O \models dp_1 \sqsubseteq dp_2$ if and only if $O_\sigma \models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$.*

Proof. ($\Leftarrow$) We prove the contrapositive: if $O \not\models dp_1 \sqsubseteq dp_2$, then $O_\sigma \not\models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$. Assume that $O \not\models dp_1 \sqsubseteq dp_2$; then a model I of O exists such that $I \not\models dp_1 \sqsubseteq dp_2$. Clearly, we have

$$dp_1 \neq \text{owl:bottomDataProperty} \quad \text{and}$$
$$dp_2 \neq \text{owl:topDataProperty}.$$

Let $\langle i, c\rangle$ be a pair of an individual and a data value contained in the interpretation of dp_1 in I, but not in the interpretation of dp_2 in I. We conservatively extend I to I' by interpreting the symbols in O_σ that do not occur in O as follows:

- the interpretation of D in I' contains only c, and
- for each $dp \in \mathbf{DP}'_O$, the interpretation of $\sigma(dp)$ in I' contains each i' such that $\langle i', c\rangle$ is contained in the interpretation of dp in I.

The interpretation I' clearly satisfies O and all axioms of the form (51) in O_σ; thus, I' is a model of O_σ. If we have dp_1 = owl:topDataProperty, then i is clearly in the interpretation of $\sigma(dp_1)$; otherwise, since $\langle i, c\rangle$ is in the interpretation of dp_1 and c is in the interpretation of D, then i is in the interpretation of $\sigma(dp_1)$ by the construction of I'. Similarly, if we have dp_2 = owl:bottomDataProperty, then i is clearly not in the interpretation of $\sigma(dp_2)$; otherwise, since $\langle i, c\rangle$ is not in the interpretation of dp_2, then i is not in the interpretation of $\sigma(dp_2)$ by the construction of I'. Consequently, $I' \not\models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$, so $O_\sigma \not\models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$, as required.

($\Rightarrow$) We consider the following cases, depending on whether dp_1 and/or dp_2 are equal to owl:topDataProperty.

1. Assume that dp_2 = owl:topDataProperty. Then we have $\sigma(dp_2)$ = owl:Thing, so $O_\sigma \models \sigma(dp_1) \sqsubseteq$ owl:Thing, and $O_\sigma \models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$, as required.
2. Assume that dp_1 = owl:topDataProperty and $dp_2 \neq dp_1$. If O is unsatisfiable, then O_σ is unsatisfiable as well, so the claim clearly holds. Assume that O is satisfiable in a

model I. Let I' be an interpretation that coincides with I on all symbols and the object domain, and whose data domain is obtained by extending the data domain of I with an arbitrary constant α. The proof of [4, Theorem DS1] shows that I' is a model of O; however, since all symbols are interpreted in I' as in I, and the interpretation of dp_1 = owl:topDataProperty contains the new constant α, it is not the case that the interpretation of dp_1 is contained in the interpretation of dp_2 in I'. Consequently, $O \not\models dp_1 \sqsubseteq dp_2$, and our claim holds vacuously.

3. In all other cases, we show the contrapositive claim: if $O_\sigma \not\models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$, then $O \not\models dp_1 \sqsubseteq dp_2$. Assume that $O_\sigma \not\models \sigma(dp_1) \sqsubseteq \sigma(dp_2)$; then a model I of O_σ exists where some i is an instance of $\sigma(dp_1)$ but not of $\sigma(dp_2)$; since O_σ contains all axioms of O, I is a model of O. Furthermore, we clearly have

$$dp_1 \neq \text{owl:bottomDataProperty} \quad \text{and}$$
$$dp_2 \neq \text{owl:topDataProperty},$$

and the case dp_1 = owl:topDataProperty is covered in Point 2, so we assume $dp_1 \neq$ owl:topDataProperty. Since we assume i to be an instance of $\sigma(dp_1)$, due to axiom (51) for dp_1 then c exists that is a dp_1-successor of i and that is an instance of D in I. But then, $O \not\models dp_1 \sqsubseteq dp_2$ holds trivially for dp_2 = owl:bottomDataProperty. Assume now that $dp_2 \neq$ owl:bottomDataProperty and that c is a dp_2-successor of i in I. Axiom (51) for dp_2 implies that i is an instance of $\sigma(dp_2)$, so i is an instance of $\sigma(dp_2)$ in I, which is a contradiction. Consequently, c is not a dp_2-successor of i—that is, $I \not\models dp_1 \sqsubseteq dp_2$—so $O \not\models dp_1 \sqsubseteq dp_2$, as required. □

An algorithm for the classification of data properties of an ontology can now be obtained by a straightforward modification of Algorithm 4.

6. Evaluation

We implemented Algorithms 1 and 4 and the adaptation of Algorithm 4 for data properties in version 1.3.5 of our HermiT reasoner. To evaluate the effectiveness of our techniques and to contrast them with the ET strategy, we compared the performance of HermiT 1.3.5 with that of HermiT 1.2.2a—an earlier version of HermiT that uses the ET algorithm for classifying both classes and properties. Both HermiT versions implement the hypertableau calculus and satisfy properties (P1) and (P2) from Section 2.3. We have *not* compared HermiT with other reasoners, as the source of any difference in performance would be difficult or impossible to determine, and so such tests would tell us very little about the effectiveness of our new classification technique. Moreover, other systems could (and we believe should) easily adopt our new technique. We conducted our evaluation using 70 well-known and widely-used ontologies. All test ontologies, both HermiT versions, the Java programs that were used to produce the results, and the test results are available online.[7]

[7] `http://www.hermit-reasoner.com/2011/clss/Evaluation.zip`

Due to lack of space, we next present the results for 20 representative ontologies on which we obtained 'interesting' results. These include two versions of the GALEN medical ontology [7],[8] several ontologies from the Open Biological Ontologies (OBO) Foundry,[9] the Food and Wine ontology from the OWL Guide, three versions of the Foundational Model of Anatomy (FMA) [6], and ontologies from the Gardiner ontology corpus [31]. For each of these ontologies, Table 1 shows the numbers of classes, properties, and assertions, as well as the fragment of OWL 2 DL that the ontology is expressed in.[10]

Each test involved classifying the classes and properties of the respective test ontology. We measured the overall classification times as well as the number of reasoning (i.e., subsumption and satisfiability) tests performed. Each classification task was performed three times and the results were averaged over the three runs. All experiments were run on an HS21 XM Blade server with two quad core Intel Xeon processors running at 2.83 GHz under 64bit Linux. We used Java 1.6, which was allowed 4GB of heap memory per test. Each test was allowed at most six hours to complete.

The results for the representative ontologies are summarised in Table 2. The upper part of the table shows the results for the deterministic ontologies (i.e., the ontologies that do not use disjunctive constructors), while the lower part shows the results for the nondeterministic ontologies. The 'data properties' columns contain '-' for ontologies without data properties, while (t/o) indicates a timeout.

As Table 2 shows, the new classification strategy of HermiT 1.3.5 is in all cases significantly faster than the ET strategy of HermiT 1.2.2a, sometimes by one or even two orders of a magnitude. This is particularly true for property classification where, as explained in Sections 4 and 5, none of HermiT's standard optimisations can be applied, and where one must entirely rely on the insertion strategy of ET to reduce the number of subsumption tests. In contrast, our reductions of property to class classification allow one to exploit all optimisations available for the classification of classes, which ensures a very good and robust performance. Note, however, that in some ontologies (e.g., NCI and biopax-level2) HermiT 1.3.5 might need roughly the same number of tests as HermiT 1.2.2a to classify the object properties. This is because in these ontologies the property hierarchy is relatively flat—that is, there are very few asserted subsumption relations between any of the object properties, so our classification algorithm performs a satisfiability test for almost all the classes that the properties are mapped to. Nonetheless, our results clearly demonstrate that correct classification of properties is practically feasible and preferable to simple but incomplete transitive closure algorithms. Finally, note that, although the Food-Wine ontology contains only one data property, the algorithms still needs to check this property

[8] We used the so-called 'doctored' (GALEN-d) and 'undoctored' (GALEN-un) versions of GALEN. Both were derived from an original GRAIL ontology, and the former is a simplified version of the latter; this simplification was necessary to allow early tableau reasoners to classify the ontology [17].

[9] `http://www.obofoundry.org/`

[10] We use the standard description logic nomenclature for fragments of OWL 2 DL [32].

Table 1: Numbers of classes, properties, and assertions, as well as the DL fragment of test ontologies

	classes	object prop.	data prop.	assertions	DL fragment
GALEN-d	2 748	413	0	0	$\mathcal{ELHIF}_{R^+}$
GALEN-un	2 748	413	0	0	$\mathcal{ELHIF}_{R^+}$
BTO	4 978	5	0	12 300	Horn-$\mathcal{ALE}$
CL-EMAPA	5 952	16	0	391	Horn-$\mathcal{ALE}$
MP	8 246	2	0	39 426	Horn-$\mathcal{AL}_{R^+}$
DOID	8 694	41	0	76 418	Horn-$\mathcal{ALH}_{R^+}$
IMR	9 164	3	0	139 447	Horn-$\mathcal{ALE}_{R^+}$
chebi	20 979	10	0	243 972	Horn-$\mathcal{ALE}_{R^+}$
NCI	27 652	70	0	0	Horn-$\mathcal{ALE}$
GO_XP	27 883	5	0	163 136	Horn-$\mathcal{SH}$
biopax-level2	42	33	37	0	$\mathcal{ALCHN}(D)$
biopax-level3	70	57	41	0	$\mathcal{SHIN}(D)$
Food-Wine	139	17	1	482	$\mathcal{SHOIN}(D)$
ProPreO	482	30	0	0	$\mathcal{SHIN}$
CL	1 498	2	0	5 908	$\mathcal{ALC}$
substance	1 721	112	33	340	$\mathcal{SHOIN}(D)$
UBERON	4 764	69	0	55 360	$\mathcal{SRI}$
FBbt_XP	7 225	21	0	12 580	$\mathcal{SHI}$
PRO	26 017	8	0	138 902	$\mathcal{SH}$
FMA 2.0-CNS	41 648	148	20	86	$\mathcal{ALCOIF}(D)$
FMA 3.0-noCNS	85 005	142	13	98	$\mathcal{ALCOIF}(D)$
FMA 3.0-noMTC	85 005	140	13	98	$\mathcal{SROIQ}(D)$

with respect to the top and bottom data property in order to insert it correctly into the hierarchy.

The results for classification of classes are similar: the new algorithm has significantly reduced the classification times in most cases. The significant performance gain in the classification of FMA is due in part to the heuristic implemented in lines 10–16 of Algorithm 2, which prevents HermiT from repeatedly testing the satisfiability of unsatisfiable classes.

Compared to the initial version of our algorithm presented in [2], our revised algorithm requires far fewer reasoning tests to classify the GALEN-d and GALEN-un ontologies. This is a consequence of identifying known subsumptions in lines 8 and 27 of Algorithm 3 even after the initialisation phase.

7. Related Work

A strategy for the construction of hierarchies that performs well for tree-like relations was described by Ellis [33]: elements are inserted into the hierarchy one at a time; furthermore, for each element, the subsumers are identified using top-down breadth-first search, and the subsumees are identified using bottom-up breadth-first search. Baader et al. [15] further refined this technique to avoid redundant subsumption tests in the top-down phase: a test $O \models A \sqsubseteq B$ is performed only if $O \models A \sqsubseteq C$ holds for each subsumer C of B [15]. Haarslev and Möller [24] further improved the traversal of flat hierarchies using a *clustering* technique, in which a single subsumption test can sometimes eliminate several potential subsumers. This technique provided us with inspiration for the efficient pruning of possible subsumers in line 6 of Algorithm 3.

Baader et al. [15] also described techniques for identifying subsumption relations between classes by analysing the syntax of ontology axioms and without running expensive subsumption tests; for example, from an axiom of the form

$$\text{SubClassOf}(A\ \text{ObjectIntersectionOf}(B\ C)) \tag{52}$$

where A, B and C are classes, one can deduce that B and C are 'told subsumers' of A. The various simplification and absorption techniques described by Horrocks [17] can increase the likelihood of identifying 'told subsumers' syntactically. Haarslev et al. [34] further extended these ideas to detect obvious non-subsumptions; for example, from axiom

$$\begin{array}{l}\text{SubClassOf}(A \\ \quad \text{ObjectIntersectionOf}(\text{ObjectComplementOf}(B)\ C))\end{array} \tag{53}$$

one can deduce that A and B are disjoint, so neither class subsumes the other (unless both are unsatisfiable). Tsarkov et al. [25] described a technique for precisely determining the subsumption relationships between 'completely defined classes'—classes whose definitions contain only conjunctions of other completely defined classes [25]. All these optimisations can be exploited in the initialisation phase of our algorithm, by suitably modifying line 1 of Algorithm 2.

8. Conclusions

In this paper, we studied the problem of efficiently classifying OWL ontologies. Unlike in earlier approaches, we consider all classification tasks, including class, object property, and data

Table 2: Evaluation results for class and property classification (time in seconds)

Ontology	Classes				Object Properties				Data Properties			
	1.2.2a		1.3.5		1.2.2a		1.3.5		1.2.2a		1.3.5	
	Tests	Time	Tests	Time	Tests	Time	Tests	Time	Tests	Time	Tests	Time
GALEN-d	2 743	3.2	2 548	1.9	5 983	358.0	196	0.2	-	-	-	-
GALEN-un	2 743	21.5	2 570	5.6	5 985	376.9	197	0.2	-	-	-	-
BTO	4 535	5.9	3 852	0.5	5	< 0.1	6	< 0.1	-	-	-	-
CL-EMAPA	5 949	15.5	3 413	0.7	16	< 0.1	17	< 0.1	-	-	-	-
MP	8 244	3.5	6 263	1.1	4	0.6	3	< 0.1	-	-	-	-
DOID	8 692	5.0	6 904	1.4	116	25.2	39	0.1	-	-	-	-
IMR	9 058	42.5	7 179	1.7	8	6.0	4	0.3	-	-	-	-
chebi	20 695	58.4	13 468	8.4	29	29.2	11	0.8	-	-	-	-
NCI	27 651	56.2	21 377	5.1	70	< 0.1	71	< 0.1	-	-	-	-
GO_XP	27 879	92.3	19 972	9.1	10	7.1	5	0.4	-	-	-	-
biopax-level2	57	< 0.1	28	< 0.1	27	0.1	33	< 0.1	1 297	2.9	45	< 0.1
biopax-level3	71	< 0.1	60	< 0.1	836	3.3	56	0.1	1 511	3.8	49	< 0.1
Food-Wine	459	17.9	191	10.2	60	10.0	11	1.3	3	0.6	2	0.2
ProPreO	1 393	5.6	1 045	5.7	341	1.6	33	< 0.1	-	-	-	-
CL	7 235	0.9	963	0.2	2	< 0.1	3	< 0.1	-	-	-	-
substance	4 725	15.6	2 918	9.8	959	19.4	108	0.3	967	18.5	38	0.1
UBERON	13 361	69.7	3 594	11.8	2 869	808.1	72	0.1	-	-	-	-
FBbt_XP	7 150	16.7	5 742	5.4	238	28.8	18	< 0.1	-	-	-	-
PRO	33 820	553.0	23 705	6.2	45	51.2	8	0.3	-	-	-	-
FMA 2.0-CNS	49 851	6 785.6	11 001	659.3	9 065	12 154.1	168	6.1	277	319.2	28	0.6
FMA 3.0-noCNS	107 280	10 200.0	23 653	2 243.5	6 291	25 400.0	116	7.1	89	345.7	22	1.2
FMA 3.0-noMTC	118 133	11 673.6	23 078	2 366.6	*t/o*	*t/o*	115	8.0	88	379.5	21	1.4

property classification. To the best of our knowledge, property classification has not previously been discussed in the literature.

We presented a new classification algorithm that significantly improves the performance of the existing class classification algorithms. The algorithm is based on the idea of maintaining two sets of known and possible subsumptions, which are updated appropriately as classification progresses. An advanced pruning strategy exploits the transitivity inherent in the subsumption hierarchy to prune these two relations and thus reduce the number of required subsumption tests.

In addition, by means of several examples, we demonstrated that commonly used algorithms for property classification based on computing the reflexive–transitive closure of the asserted property hierarchy are incomplete even for very weak fragments of OWL. Furthermore, we discussed the difficulties of applying our classification approach directly to property classification. Finally, we showed how to reduce the problems of classifying object and data properties to the problem of classifying classes. These reductions can be used to classify the property hierarchies while reusing all available optimisations.

We implemented all our algorithms in version 1.3.5 of the HermiT reasoner, and we compared the performance of HermiT 1.3.5 with an earlier version of HermiT that uses the standard enhanced traversal classification algorithm. Our results are very encouraging, showing significant improvements in classification times and reductions in the number of subsumption tests. Our experiments also show that both correct and efficient classification of object and data properties is possible in practice.

We are currently working on extending our algorithm to *individual realisation*—the tasks of computing, for each individual i in an ontology, the most specific classes C such that i is an instance of C. Our preliminary results suggest that the performance of realisation can also be significantly improved by exploiting the ideas outlined in this paper.

Acknowledgements

The research presented in this paper was funded by the EPSRC project HermiT: Reasoning with Large Ontologies. The evaluation has been performed on computers of the Baden-Württemberg Grid (the bwGRiD project[11]), member of the German D-Grid initiative, funded by the Ministry of Education and Research (Bundesministerium fuer Bildung und Forschung) and the Ministry of Science, Research and the Arts Baden-Wuerttemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg). Birte Glimm acknowledges the support of the Transregional Collaborative Research Center SFB/TRR 62 “Companion-Technology for Cognitive Technical Systems” funded by the German Research Foundation (DFG).

References

[1] R. Shearer, I. Horrocks, Exploiting partial information in taxonomy construction, in: Proc. of the 8th International Semantic Web Conference (ISWC 2009), volume 5823 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 569–584.

[2] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Optimising ontology classification, in: Proc. of the 9th International Semantic Web Conference (ISWC 2010), volume 6496 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 225–240.

[3] B. Motik, P. F. Patel-Schneider, B. Parsia, OWL 2 Web Ontology Language Structural Specification and Functional-style Syntax, W3C Recommendation, 2009.

[4] B. Motik, P. F. Patel-Schneider, B. Cuenca Grau, OWL 2 Web Ontology Language Direct Semantics, W3C Recommendation, 2009.

[5] A. Sidhu, T. Dillon, E. Chang, B. S. Sidhu, Protein ontology development using OWL, in: Proc. of the OWL: Experiences and Directions Workshop (OWLED 2005), volume 188 of *CEUR Workshop Proceedings*, Galway, Ireland.

[11] `http://www.bw-grid.de`

[6] C. Golbreich, S. Zhang, O. Bodenreider, The foundational model of anatomy in OWL: Experience and perspectives, Journal of Web Semantics 4 (2006) 181–195.
[7] A. Rector, J. Rogers, Ontological and practical issues in using a description logic to represent medical concept systems: Experience from GALEN, in: Reasoning Web, volume 4126 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 197–231.
[8] D. Soergel, B. Lauser, A. Liang, F. Fisseha, J. Keizer, S. Katz, Reengineering thesauri for new applications: The agrovoc example, Journal of Digital Information 4 (2004).
[9] S. Derriere, A. Richard, A. Preite-Martinez, An ontology of astronomical object types for the virtual observatory, in: Proc. of the 26th meeting of the IAU: Virtual Observatory in Action: New Science, New Technology, and Next Generation Facilities, Prague, Czech Republic, pp. 17–18.
[10] L. Lacy, G. Aviles, K. Fraser, W. Gerber, A. Mulvehill, R. Gaskill, Experiences using OWL in military applications, in: Proc. of the OWL: Experiences and Directions Workshop (OWLED 2005), volume 188 of *CEUR Workshop Proceedings*, Galway, Ireland.
[11] J. Goodwin, Experiences of using OWL at the ordnance survey, in: Proc. of the OWL: Experiences and Directions Workshop (OWLED 2005), volume 188 of *CEUR Workshop Proceedings*, Springer, Galway, Ireland, 2005.
[12] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL DL reasoner, Journal of Web Semantics 5 (2007) 51–53.
[13] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: System description, in: Proc. of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), volume 4130 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 292–297.
[14] V. Haarslev, R. Möller, Racer system description, in: Proc. of the 1st International Joint Conference on Automated Reasoning (IJCAR 01), volume 2083 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 701–706.
[15] F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, E. Franconi, An empirical analysis of optimization techniques for terminological representation systems, Applied Intelligence 4 (1994) 109–132.
[16] I. Horrocks, P. F. Patel-Schneider, Optimising description logic subsumption, Journal of Logic and Computation 9 (1999) 9–3.
[17] I. Horrocks, Optimising Tableaux Decision Procedures for Description Logics, Ph.D. thesis, University of Manchester, 1997.
[18] D. Tsarkov, I. Horrocks, Ordering heuristics for description logic reasoning, in: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Morgan Kaufmann, 2005, pp. 609–614.
[19] Y. Ding, V. Haarslev, Tableau caching for description logics with inverse and transitive roles, in: Proc. of the 2006 Description Logic Workshop, volume 189 of *CEUR Workshop Proceedings*.
[20] E. Sirin, B. Cuenca Grau, B. Parsia, From wine to water: Optimizing description logic reasoning for nominals, in: Proc. of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), The AAAI Press, 2006, pp. 90–99.
[21] B. Glimm, I. Horrocks, B. Motik, Optimized description logic reasoning via core blocking, in: Proc. of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010), volume 6173 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 457–471.
[22] F. Baader, S. Brandt, C. Lutz, Pushing the $\mathcal{EL}$ envelope, in: Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05), Morgan Kaufmann Publishers Inc., 2005, pp. 364–369.
[23] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz, OWL 2 Web Ontology Language Profiles, W3C Recommendation, 2009.
[24] V. Haarslev, R. Möller, High performance reasoning with very large knowledge bases: A practical case study, in: Proc. of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01), Morgan Kaufmann, 2001, pp. 161–168.
[25] D. Tsarkov, I. Horrocks, P. F. Patel-Schneider, Optimizing terminological reasoning for expressive description logics, Journal of Automated Reasoning 39 (2007) 277–316.
[26] B. Motik, R. Shearer, I. Horrocks, Hypertableau reasoning for description logics, Journal of Artificial Intelligence Research 36 (2009) 165–228.
[27] I. Horrocks, O. Kutz, U. Sattler, The even more irresistible $\mathcal{SROIQ}$, in: Proc. of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006), The AAAI Press, 2006, pp. 57–67.
[28] J. Y. Halpern, Y. Moses, A guide to completeness and complexity for model logics of knowledge and belief, Artificial Intelligence 54 (1992) 319–379.
[29] Y. Kazakov, B. Motik, A resolution-based decision procedure for $\mathcal{SHOIQ}$, Journal of Automated Reasoning 40 (2008) 89–116.
[30] B. Motik, I. Horrocks, OWL datatypes: Design and implementation, in: Proc. of the 7th International Semantic Web Conference (ISWC 2008), volume 5318 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 307–322.
[31] T. Gardiner, D. Tsarkov, I. Horrocks, Framework for an automated comparison of description logic reasoners, in: Proc. of the 5th International Semantic Web Conference (ISWC 2006), volume 4273 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 654–667.
[32] F. Baader, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, implementation and applications, Cambridge University Press, 2002.
[33] G. Ellis, Compiled hierarchical retrieval, in: T. Nagle, J. Nagle, L. Gerholz, P. Eklund (Eds.), Conceptual Structures: Current Research and Practice, Ellis Horwood, 1992, pp. 271–294.
[34] V. Haarslev, R. Möller, A.-Y. Turhan, Exploiting pseudo models for TBox and ABox reasoning in expressive description logics, in: Proc. of the 1st International Joint Conference on Automated Reasoning (IJCAR 01), volume 2083 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 61–75.