Noname manuscript No.
(will be inserted by the editor)

HermiT: An OWL 2 Reasoner

Birte Glimm · Ian Horrocks · Boris Motik · Giorgos Stoilos · Zhe Wang

Received: date / Accepted: date

Abstract This system description paper introduces the OWL 2 reasoner HermiT. The reasoner is fully compliant with the OWL 2 Direct Semantics as standardised by the World Wide Web Consortium (W3C). HermiT is based on the hypertableau calculus, and it supports a wide range of standard and novel optimisations to improve the performance of reasoning on real-world ontologies. Apart from the standard OWL 2 reasoning task of entailment checking, HermiT supports several specialised reasoning services such as class and property classification, as well as a range of features outside the OWL 2 standard such as DL-safe rules, SPARQL queries, and description graphs. We discuss the system's architecture, and we present an overview of the techniques used to support the mentioned reasoning tasks. We further compare the performance of reasoning in HermiT with that of FaCT++ and Pellet—two other popular and widely used OWL 2 reasoners.

Birte Glimm
University of Ulm, Institute of Artificial Intelligence,
89069 Ulm, DE
E-mail: birte.glimm@uni-ulm.de

Ian Horrocks
University of Oxford, Department of Computer Science,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
E-mail: ian.horrocks@cs.ox.ac.uk

Boris Motik
University of Oxford, Department of Computer Science,
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
E-mail: boris.motik@cs.ox.ac.uk

Giorgos Stoilos
National Technical University of Athens, School of Electrical & Computer Engineering
Iroon Polytexneiou 9, 15780 Zografou, GR
E-mail: gstoil@image.ntua.gr

Zhe Wang
Griffith University, School of Information & Communication Technology
Brisbane, QLD 4111, AU
E-mail: zhe.wang@griffith.edu.au

1 Introduction

In this system description paper we describe the main features of the HermiT ontology reasoner. HermiT supports all features of the OWL 2 ontology language [3], including all OWL 2 datatypes [23], and it correctly performs both object and data property classification—reasoning tasks that are, to the best of our knowledge, not fully supported by any other OWL reasoner. In addition to these standard reasoning tasks, HermiT also supports SPARQL query answering, and it uses a range of optimisations [18] to ensure efficient processing of real-world ontologies. Furthermore, HermiT supports several features that go beyond existing standards, such as DL-safe SWRL rules [14,26] and description graphs [21]—an extension of OWL 2 that allows for a faithful modelling of arbitrarily connected structures.

A key novel idea in HermiT is the hypertableau calculus [27], which allows the reasoner to avoid some of the nondeterministic behaviour exhibited by the tableau calculus used in Pellet [33] and FaCT++ [35]—two other popular and widely used OWL reasoners. In order to further improve the performance of the calculus, HermiT employs a wide range of standard and novel optimisation techniques, including anywhere blocking [27], blocking signature caching [27], individual reuse [22], and core blocking [6], and it uses various heuristics to automatically select from amongst a range of different optimisation strategies. HermiT also implements a novel classification algorithm [7] that greatly reduces the number of consistency tests needed to compute the class and property hierarchies.

We have compared HermiT's performance with Pellet [33] and FaCT++ [35] on a set of standard ontologies. Unlike many earlier evaluations, all the ontologies used in our tests are directly accessible via immutable URIs, so our tests are fully repeatable and their results are unambiguous. The results show that, although HermiT did not outperform the other reasoners on all ontologies, it seemed more robust as it managed to process more 'hard' ontologies.

The rest of this paper is organised as follows: in Section 2 we introduce HermiT's system architecture; in Section 3 we present an overview of the hypertableau calculus; in Section 4 we discuss several optimisations of the core calculus; in Section 5 we outline the features that go beyond OWL 2 and discuss their support in the reasoner; and in Section 6 we evaluate HermiT's performance on a wide range of ontologies and compare it to the performance of Pellet and FaCT++.

We assume the reader to be familiar with OWL, description logics [19,11], and the correspondence between OWL syntax and description logic syntax [13]. We take an OWL ontology $\mathcal{O}$ to consist of a *TBox* $\mathcal{T}$ and an *ABox* $\mathcal{A}$, where the former specifies the schema (i.e., the axioms that describe the structure of the domain being modelled) and the later contains the data (i.e., the assertions describing the objects in a domain of discourse). Although HermiT is an OWL reasoner, for brevity we will mainly use the standard description logic syntax. Furthermore, we will talk about *classes* and *properties*, which are commonly called *concepts* and *roles* in the description logic community.

2 System Architecture

HermiT consists of several components that together implement a sound and complete OWL reasoning system. Figure 1 shows the most important components (e.g.,

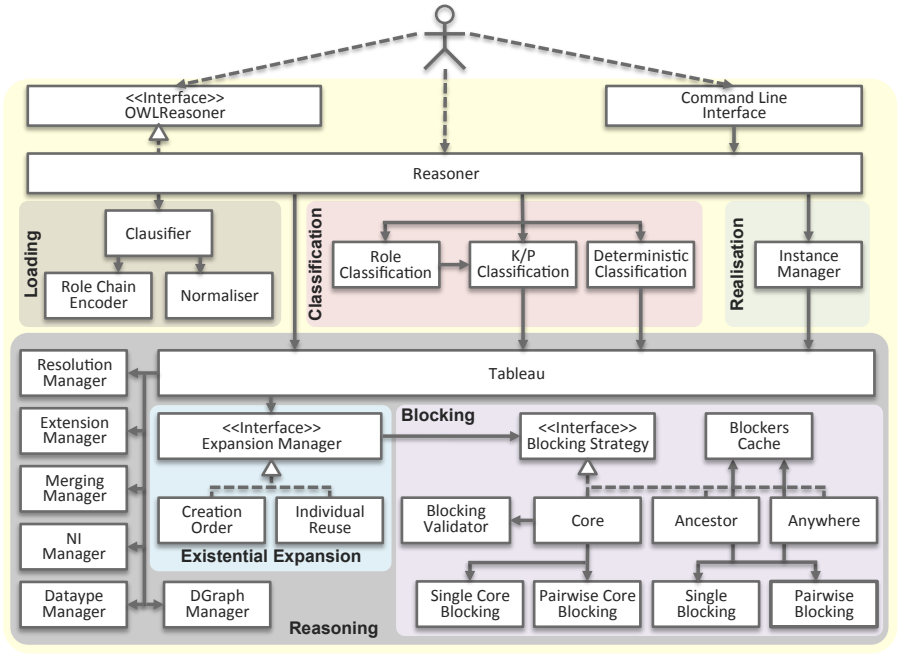


Fig. 1: A diagram showing the different components of the HermiT reasoning system

`Loading`, `Classification`, etc.) and their main subcomponents. The system has been implemented in Java for portability and easy integration into applications.

Users can interact with the reasoner via three different interfaces: a native Java interface, the OWL API [12], and a command line interface. HermiT's native interface (the `Reasoner` component) is a facade that converts typical reasoning tasks into ontology consistency tests—the only reasoning operation supported by the hypertableau calculus. For example, to check whether the currently loaded ontology entails that an object property f is functional, the `Reasoner` component temporarily extends the ontology's ABox with two object property assertions that relate a fresh individual a via the f property with two fresh individuals b and c, and an assertion specifying that b and c are distinct individuals, and then it checks whether such an extended ontlogy is consistent using the hypertableau calculus; if that is the case, then a model exists proving f to be not functional.

The `Reasoner` component also implements the `OWLReasoner` interface from the OWL API. This allows HermiT to be used in any application based on the OWL API, and it also allows the Protégé editor to use HermiT as a plugin. HermiT does not internally use the OWL API data structures to represent ontologies and axioms, so the `Reasoner` component converts OWL API data structures into HermiT's internal data structures and back.

The command line interface allows users to invoke basic reasoning tasks from the command line. In order to keep the number of command line options manageable, the interface does not expose all of the inferencing capabilities of the reasoner: only common tasks such as ontology classification are supported. The

main benefit of the command line interface is that it allows HermiT to be used without any prior setup (e.g., writing a Java program that invokes the `Reasoner` component or the `OWLReasoner` interface).

2.1 Loading an Ontology

HermiT internally represents an ontology as a set of (ground) assertions $\mathcal{A}$ and a set of *DL-clauses* $\mathcal{C}$. A DL-clause is an implication of the form

$$B_1 \wedge \ldots \wedge B_n \rightarrow H_1 \vee \ldots \vee H_m \tag{1}$$

where each B_i is an atom (i.e., an assertion of the form $A(x)$ with A a class or $R(x, y)$ with R a property), and each H_j is an atom or an expression of the form $\exists R.A(x)$. A DL-clause straightforwardly corresponds to a first-order implication.

The `Reasoner` component is given an OWL ontology, and the task of the `Loading` component is construct the set $\mathcal{O}$ of all axioms contained in the given and all directly and indirectly imported ontologies,[1] and then to convert $\mathcal{O}$ into sets $\mathcal{A}$ and $\mathcal{C}$. Towards this goal, the `Normaliser` component first simplifies complex axioms (e.g., by removing duplicate or irrelevant conjuncts or disjuncts) and then converts the result into a particular normal form; the normalisation step can be seen as a variant of the well-known structural transformation [29,28]. For example, the complex superclass in the axiom

$$\mathit{Person} \sqsubseteq \forall \mathit{hasAncestor}.(\mathit{Male} \sqcup \mathit{Female}) \tag{2}$$

is normalised so that only a fresh class Q occurs inside the quantifier, and the meaning of Q is captured in a separate axiom, si the above axiom is transformed into the following two axioms:

$$\mathit{Person} \sqsubseteq \forall \mathit{hasAncestor}.Q \tag{3}$$

$$Q \sqsubseteq \mathit{Male} \sqcup \mathit{Female} \tag{4}$$

ABox assertions are subjected to the same transformation so, after normalisation, the ABox contains only classes and properties, rather than class expressions and property expressions. Moreover, if present in the input ontology, SWRL rules are normalised along the same lines.

OWL supports transitive object properties and property chain axioms; however, these features are not handled directly by the the hypertableau calculus as this would make it difficult to ensure termination of the calculus. Instead, the `Role Chain Encoder` component transforms away transitivity and property chain axioms by introducing additional axioms that ensure equisatisfiability between the original and the transformed ontologies; the transformation is based on an automata-based technique [32]. The `Role Chain Encoder` component also introduces axioms encoding the semantics of the the universal object property (i.e., *owl:topObjectProperty*) [20] if needed (i.e., if it occurs in the ontology).

[1] OWL ontologies can include references to other ontologies that are to be 'imported' (syntactically added) into the current ontology [25].

The resulting axioms are finally converted directly into DL-clauses. For example, the normalised axioms (3) and (4) are translated into the following DL-clauses:

$$Person(x) \wedge hasAncestor(x, y) \rightarrow Q(y) \tag{5}$$

$$Q(x) \rightarrow Male(x) \vee Female(x) \tag{6}$$

During the loading step, the ontology expressivity is also determined (e.g., whether the ontology contains inverse properties or nominals), and this information is subsequently used to automatically configure certain options, such as how to parameterise the hypertableau calculus.

2.2 Reasoning-Related Components

As already mentioned, all reasoning tasks are transformed into one or more ontology consistency tests, whose goal is to determine the consistency of the currently loaded sets of assertions $\mathcal{A}$ and DL-clauses $\mathcal{C}$. For certain reasoning tasks (e.g., to determine subsumption between complex class expressions), it might be necessary to extend $\mathcal{A}$ and $\mathcal{C}$ with temporary assertions and DL-clauses that are needed only for this single reasoning task. The satisfiability of $\mathcal{A}$ and $\mathcal{C}$ is decided using the hypertableau calculus [27], which tries to construct a *pre-model*—a finite set of (ground) assertions that describe a (possibly infinite) model satisfying $\mathcal{A}$ and $\mathcal{C}$. The `Reasoning` component implements the hypertableau calculus, and we discuss both the calculus and its implementation in more detail in Section 3.

The `Classification` component uses the `Reasoning` component to compute class and property hierarchies—that is, to arrange all classes, object properties, and data properties occurring in the ontology into hierarchies that correctly reflect the relevant subsumption relationships. To classify classes, HermiT uses a novel algorithm [7] that extracts information from the constructed pre-models in order to reduce the number of subsumption tests performed. To the best of our knowledge, all reasoners apart from HermiT classify object and data properties by simply computing the reflexive–transitive closure of the asserted property inclusion axioms, which is known to be incomplete (for both object and data properties) even in very simple ontology languages [7]. In contrast, HermiT reduces property classification to class classification [7], and so it is the only reasoner that is guaranteed to be complete for object and data property classification.

The `Realisation` component uses the `Reasoning` component to compute the set of instances for each class and property in the ontology. Similarly to classification, HermiT optimises this computation by exploiting pre-models constructed during each consistency test. In its default mode, HermiT supports *lazy realisation*, where known and possible instances of classes are initialised during an initial ontology consistency test, and refined during subsequent query answering. One can, however, also explicitly request a complete computation of the instances of all classes and properties.

3 Hypertableau Reasoning in HermiT

In this section we describe the `Reasoning` component of HermiT, which implements the core reasoning task of deciding consistency of a set of assertions $\mathcal{A}$ and a set

of DL-clauses $\mathcal{C}$. In particular, in Section 3.1 we describe the calculus from a conceptual perspective, in Section 3.2 we present a small example and contrast the calculus with the known tableau calculi, and in Section 3.3 we discuss how the calculus was implemented within the HermiT's architecture from Figure 1.

3.1 The Hypertableau Calculus

The formal definition of the hypertableau calculus [27] is rather involved, so we do not repeat it here; rather, we discuss the calculus informally, and in Section 3.2 we illustrate some of its key aspects. The calculus takes as input an ABox $\mathcal{A}$ and a set of DL-clauses $\mathcal{C}$. Each DL-clause in $\mathcal{C}$ must be of the form

$$\bigwedge A_i(x) \wedge \bigwedge p_i(x, y_i) \wedge \bigwedge B_i(y_i) \wedge \bigwedge O_{a_i}(z_{a_i}) \rightarrow \\ \bigvee C_i(x) \vee \bigvee D_i(y_i) \vee \bigvee r_i(x, y_i) \vee \bigvee x \approx z_{a_i} \vee \bigvee y_i \approx y_j @^x_{\leq n\, p.C},$$

where p_i and r_i are object or data property expressions, A_i and B_i are classes, and C_i and D_i are classes, possibly negated datatypes, enumerations of data values, or class expressions of the form $\geq n\, p.A$ or $\leq n\, p.\neg A$. Atoms of the form $x \approx z_{a_i}$ stem from nominals; for example, axiom $C \sqsubseteq \{a\}$ is translated into a DL-clause $C(x) \wedge O_a(z_a) \rightarrow x \approx z_a$ and an assertion $O_a(a)$, where O_a is a new class uniquely associated with the nominal $\{a\}$. Finally, equalities of the form $y_i \approx y_j @^x_{\leq n\, p.C}$ stem from at-most number restrictions; for example, $\top \sqsubseteq \leq 1\, p.\top$ with p an object property is translated into a DL-clause $p(x, y_1) \wedge p(x, y_2) \rightarrow y_1 \approx y_2 @^x_{\leq 1\, p.\top}$. An expression $@^x_{\leq n\, p.C}$ is called an *annotation* and it essentially captures the at-most number restriction (i.e., $\leq n\, p.C$ in this case) that gives rise to the equality; this allows us to correctly handle nominals in the presence of inverse properties and number restrictions (cf. the nominal introduction rule described below). Annotations are not needed for at-most restrictions over data properties.

The hypertableau calculus solves the consistency problem: given a set of DL-clauses $\mathcal{C}$ and an ABox $\mathcal{A}$, it determines whether $\mathcal{C} \cup \mathcal{A}$ is consistent. This is achieved by constructing a *derivation* for $\mathcal{A}$ w.r.t. $\mathcal{C}$, which is a sequence of sets of assertions $\mathcal{A}_0, \ldots, \mathcal{A}_n$ where

- $\mathcal{A}_0 = \mathcal{A}$,
- for each $0 \leq \ell < n$, the ABox $A_{\ell+1}$ is a possible result of applying a *derivation rule* to $\mathcal{A}_\ell$, and
- no derivation rule is applicable to $\mathcal{A}_n$.

If a derivation for $\mathcal{A}$ w.r.t. $\mathcal{C}$ exists such that $\mathcal{A}_n$ does not contain an obvious contradiction (called a *clash*), then $\mathcal{C} \cup \mathcal{A}$ is consistent and $\mathcal{A}_n$ is called a *pre-model* of $\mathcal{C} \cup \mathcal{A}$. In contrast, if no such derivation can be constructed, then $\mathcal{C} \cup \mathcal{A}$ is inconsistent. Derivation construction is nondeterministic: there can be more than one way of constructing $\mathcal{A}_{i+1}$ from $\mathcal{A}_i$. Thus, to determine that a pre-model of $\mathcal{C} \cup \mathcal{A}$ does not exist, all nondeterministic choices must be explored, and this is commonly achieved via backtracking. In the rest of this section we present an overview of the inference rules used to construct a derivation for $\mathcal{A}$ w.r.t. $\mathcal{C}$.

The main derivation rule is the *Hyp-rule*, which ensures that a pre-model satisfies all consequences of the DL-clases in $\mathcal{C}$. Let σ be a mapping from variables to individuals, and let U be an atom; then, $\sigma(U)$ is the atom obtained from U by replacing each variable x with $\sigma(x)$. The Hyp-rule is applicable to a DL-clause

$\bigwedge_{i=1}^{m} U_i \rightarrow \bigvee_{j=1}^{n} V_j$ in $\mathcal{C}$ and an ABox $\mathcal{A}_\ell$ if a mapping σ from the variables in the DL-clause to the individuals in $\mathcal{A}_\ell$ exists such that $\sigma(U_i) \in A_\ell$ for each $1 \leq i \leq m$ and $\sigma(V_j) \notin \mathcal{A}_\ell$ for each $1 \leq j \leq n$. If this is the case, then the rule nondeterministically derives $\mathcal{A}_{\ell+1} = \mathcal{A}_\ell \cup \{\sigma(V_j)\}$ for some $1 \leq j \leq n$. For example, applying the DL-clause $A(x) \rightarrow (\geq 1\, p.B)(x) \vee C(x)$ to ABox $\mathcal{A}_\ell$ containing assertion $A(s)$ extends $\mathcal{A}_\ell$ with either $(\geq 1\, p.B)(s)$ or $C(s)$.

The *$\geq$-rule* ensures that a pre-model satisfies all at-least restrictions. In particular, the rule is applicable to $\geq n\, p.C(s)$ in $\mathcal{A}_\ell$ if and only if $\mathcal{A}_\ell$ does not contain individuals $u_1, \ldots, u_n$ such that $p(s, u_i) \in A_\ell$ and $C(u_i) \in \mathcal{A}_\ell$ for each $1 \leq i \leq n$, and $u_i \not\approx u_j \in \mathcal{A}_\ell$ for each $1 \leq i < j \leq n$. If that is the case, then the rule deterministically derives

$$\mathcal{A}_{\ell+1} = \mathcal{A}_\ell \cup \{p(s, t_i), C(t_i) \mid 1 \leq i \leq n\} \cup \{t_i \not\approx t_j \mid 1 \leq i < j \leq n\},$$

where $t_1, \ldots, t_n$ are fresh individuals. Individual s is called a *predecessor* of each individual t_i, and each t_i is a called a *successor* of s; moreover, *ancestor* and *descendant* are transitive closures of the predecessor and successor relations, respectively. The $\geq$-rule introduces fresh individuals, so an unrestricted application of the rule can easily lead to the introduction of an infinite number of fresh individuals and thus prevent the calculus from terminating. In order to ensure termination, the hypertableau calculus uses *blocking*, which in turn uses the notion of a *label* of an individual or an individual pair. In particular, let s and t be individuals and let $\mathcal{A}$ be an ABox; then, $\mathcal{L}_\mathcal{A}(s)$ and $\mathcal{L}_\mathcal{A}(s, t)$ are defined as follows:

$$\mathcal{L}_\mathcal{A}(s) = \{B \mid B(s) \in \mathcal{A}\} \qquad \mathcal{L}_\mathcal{A}(s, t) = \{r \mid r(s, t) \in \mathcal{A}\}$$

In other words, $\mathcal{L}_\mathcal{A}(s)$ is the set of classes that label s in $\mathcal{A}$, and $\mathcal{L}_\mathcal{A}(s, t)$ is the set of atomic roles that connect s and t in $\mathcal{A}$. The hypertableau calculus can be used with two distinct blocking variants. *Single* blocking is applicable to ontologies that do not contain inverse properties: an individual s is *directly single-blocked* in $\mathcal{A}$ by an individual s' if $\mathcal{L}_\mathcal{A}(s) = \mathcal{L}_\mathcal{A}(s')$. If an ontology contains inverse properties, then *pairwise* blocking is needed: an individual s with predecessor t is *directly pairwise-blocked* in $\mathcal{A}$ by an individual s' with predecessor t' if $\mathcal{L}_\mathcal{A}(s) = \mathcal{L}_\mathcal{A}(s')$, $\mathcal{L}_\mathcal{A}(t) = \mathcal{L}_\mathcal{A}(t')$, $\mathcal{L}_\mathcal{A}(s, t) = \mathcal{L}_\mathcal{A}(s', t')$, and $\mathcal{L}_\mathcal{A}(t, s) = \mathcal{L}_\mathcal{A}(t', s')$. In both cases, care must be taken to avoid cyclic blocks; we omit the details for brevity. An individual s is *blocked* in $\mathcal{A}$ if either s or some of its ancestors is directly blocked in $\mathcal{A}$ by another individual. The $\geq$-rule is applicable to an assertion $\geq n\, p.C(s)$ only if s is not blocked, which ensures termination without affecting calculus' completeness. This can be intuitively understood as follows. Due to the restricted shape of DL-clauses, each ABox $\mathcal{A}$ encountered in a derivation has a certain forest shape. Thus, if an individual s is directly blocked by an individual s' in $\mathcal{A}$ and if both individuals occur in the tree part of $\mathcal{A}$, then the two individuals 'behave' in the same way: all inferences that can be applied to s' can be applied to s as well. Thus, instead of applying the $\geq$-rule to $\geq n\, p.C(s)$ to create an appropriate subtree, we can 'copy-and-paste' the subtree of s' and thus satisfy the assertion. We discuss this in more detail in Section 3.2 by means of an example.

The *$\approx$-rule* deals with equality assertions. In particular, the $\approx$-rule is applicable to an assertion $s \approx t$ in $\mathcal{A}_\ell$ if $s \neq t$, and an application of the $\approx$-rule merges s and t, as we describe next. To address termination problems caused by merging, the $\approx$-rule always merges a descendent into its ancestor, and a named individual only into

another named individual. Moreover, when individual s is merged into individual t, then s is first *pruned*—that is, all assertions involving a descendant of s are removed; next, s is replaced with t in all assertions that contain s.

The *nominal introduction rule* (*NI-rule*) deals with equalities introduced by number restrictions of the form $\leq n\, p.C$. Interactions between nominals and inverse properties can give rise to ABoxes that are not forest-shaped, which can prevent blocking and thus cause termination problems. The NI-rule deals with these problems by promoting some of the individuals introduced by the $\geq$-rule into *root individuals*—that is, the promoted individuals are treated in the same way as the named individuals in $\mathcal{A}_0$, so they may be arbitrarily interconnected. The NI-rule is applicable to an assertion $s \approx t@^{u}_{\leq n\, p.C}$ in $\mathcal{A}_\ell$ if u is a root individual, s and t are not root individuals, and s is not a successor of u. If that is the case, then the NI-rule nondeterministically merges s into a root individual of a special form that was designed so that the number of such individuals can be bounded by the size of $\mathcal{A}'$ and $\mathcal{C}$. Furthermore, to ensure termination, the NI-rule must be applied before the $\approx$-rule—that is, if both rules are applicable to the same equality assertion, the NI-rule should be applied first. Finally, the NI-rule is necessary only if nominals, inverse properties, and number restrictions are all used together in (possibly different) ontology axioms.

Finally, the $\bot$-*rule* detects obvious contradictions, called *clashes*. The rule is applicable to an assertion $s \not\approx s$, or to assertions $A(s)$ and $\neg A(s)$ in $\mathcal{A}_\ell$. If that is the case, then the rule derives $\mathcal{A}_{\ell+1} = \mathcal{A}_\ell \cup \{\bot\}$, which captures the information that ABox $\mathcal{A}_{\ell+1}$ contains a contradiction.

3.2 An Example

We next discuss certain aspects of the hypertableau calculus using an ontology $\mathcal{O}_1$ containing assertion (7), and axioms (8) instantiated for each $0 \leq i \leq n$.

$$A_0(a) \tag{7}$$

$$A_i \sqsubseteq \exists r_i.A_{(i+1) \bmod (n+1)} \sqcap \exists s_i.A_{(i+1) \bmod (n+1)} \tag{8}$$

To apply the hypertableau calculus, these axioms are first preprocessed as described in Section 2.1, so axiom (8) is converted to to DL-clauses (9) and (10).

$$A_i(x) \rightarrow \exists r_i.A_{(i+1) \bmod (n+1)}(x) \tag{9}$$

$$A_i(x) \rightarrow \exists s_i.A_{(i+1) \bmod (n+1)}(x) \tag{10}$$

It should be clear that applying the *Hyp*-rule and the $\geq$-rule without blocking produces an infinite tree-shaped ABox shown in Figure 2; in the figure, individuals are represented as nodes, property assertions between individuals are represented as arcs, and class assertions are represented by labelling individuals with class expressions. The ABox is infinite because implications between existential quantifiers in $\mathcal{O}_1$ are cyclic. Note, however, that each individual s at the bottom of the figure labelled with A_0 'behaves' in the same way as individual a: the labels of the two individuals coincide, so s is blocked by a. Each inference applicable to s is a 'copy' of an inference applicable to a; thus, even if we do not apply the $\geq$-rule to assertions $\exists r_i.A_{(i+1) \bmod (n+1)}(s)$ and $\exists s_i.A_{(i+1) \bmod (n+1)}(s)$, we can satisfy the

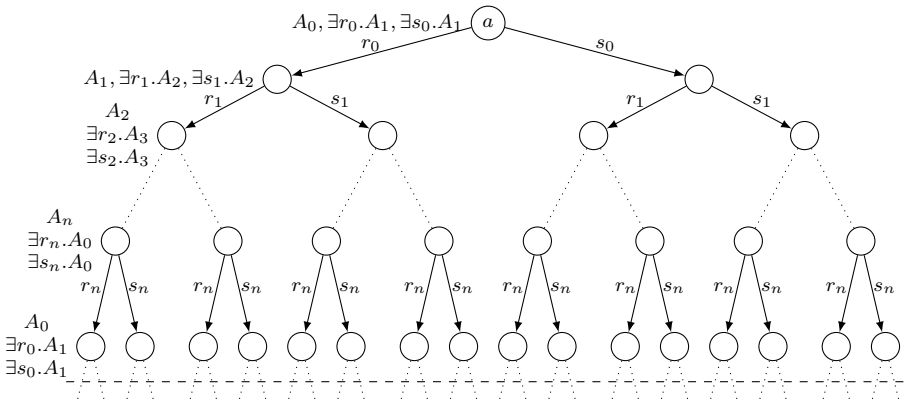


Fig. 2: Infinite Tree Axiomatised by the Example Ontology

assertions by 'copying' the subtree of a. Hence, the finite part of the ABox above the dashed line is a pre-model for assertion (7) and DL-clauses (9)–(10).

We next compare the hypertableau calculus with tableau calculi. To this end, let $\mathcal{O}_2$ be $\mathcal{O}_1$ extended with axiom (11), which is converted to DL-clause (12).

$$\exists r_0.\neg B \sqsubseteq B \tag{11}$$

$$r_0(x, y) \rightarrow B(x) \vee B(y) \tag{12}$$

Tableau calculi [2] are similar in principle to the hypertableau calculus: they are given an ontology consisting of an ABox $\mathcal{A}$ and a TBox $\mathcal{T}$, and they construct a derivation $\mathcal{A}_0, \ldots, \mathcal{A}_n$ for $\mathcal{A}$ w.r.t. $\mathcal{T}$ using various inference rules. The main difference to the hypertableau calculus is in the treatment of class inclusion axioms. At least in their unoptimised form [2], tableau calculi transform such axioms into negation-normal form and then apply them to all individuals occurring in a derivation. For example, an axiom α of the form $A \sqcap B \sqsubseteq C$ is converted into an axiom $\top \sqsubseteq \neg A \sqcup \neg B \sqcup C$; then, to ensure that each individual s occurring in a derivation satisfies α, a tableau calculus nondeterministically guesses whether s satisfies $\neg A$, $\neg B$, or C. The number of such guesses is polynomial in the number of individuals occurring in a derivation and in the number of axioms of the mentioned form, which can cause problems in practice if either of these numbers is large. Tableau calculi address this problem to an extent using various *absorption* optimisations [36]. For example, the mentioned axiom α might be transformed into axiom $A \sqsubseteq \neg B \sqcup C$; then, the axiom is applied to an individual s only if s satisfies A, in which case the rule nondeterministically derives that s satisfies either $\neg B$ or C. This, however, does not eliminate nondeterminism completely, even though the original axiom is deterministic: it corresponds to the Horn clause $A(x) \wedge B(x) \rightarrow C(x)$. More advanced versions of absorption, such as *role* [34] and *binary* [16] absorption, can further reduce the degree of nondeterminism in tableau calculi. None of the known absorption techniques, however, can handle axiom (11); thus, tableau calculi derive a disjunction $\forall r_0.B \sqcup B$ for each node of the tree, which gives rise to an exponential number of nondeterministic choices.

The aim of the hypertableau calculus is to reduce the degree of nondeterminism. In our example, the *Hyp*-rule applies the DL-clause (12) only to arcs in the

tree connected by property r_0, which gives rise to a considerably smaller number of nondeterministic choices. The hypertableau calculus thus generalises standard absorption optimisations [1, Chapter 9], role absorption [34], and binary absorption [16], as well as allowing additional types of 'absorption' that are not possible in standard tableau calculi. In particular, on Horn ontologies [17]—that is, on ontologies that can be transformed into an equivalent set of Horn clauses—the calculus becomes deterministic. This is important ontologies typically considered in practice often mostly consist of Horn axioms.

3.3 Implementing the Hypertableau Calculus in HermiT

We next discuss how we implemented the calculus within the `Reasoning` component of HermiT show in Figure 1. Although the hypertableau calculus comprises relatively simple rules, implementing them efficiently is not trivial.

The `Extension Manager` component keeps track of the current set of assertions, and it also detects when this set contains a clash. All class assertions are kept in a binary table, and an assertion of the form $A(b)$ is stored in the table as a tuple $\langle A, b\rangle$; furthermore, all property assertions are kept in a ternary table, and an assertion of the form $r(a, b)$ is stored in the table as a tuple $\langle r, a, b\rangle$. Both tables are indexed so as to allow for easy retrieval of assertions. In order to facilitate backtracking, the class and property assertion tables are used as stacks: assertions are always added at the end of the table, and they are popped off the end during backtracking; moreover, no assertion is ever modified in-place. The state of the tables before a nondeterministic choice is thus fully described by two integers pointing to the tables' ends. All other components of HermiT with state that evolves over time use a similar approach, and so the state of the reasoner at any point in a derivation can be fully described by a handful of integers. This makes the introduction of nondeterministic choices very cheap, which is important since some optimisations used in HermiT (such as individual reuse [22]) can introduce a very large number of such choices.

The `Resolution Manager` component implements the Hyp-rule: for each DL-clause it attempts to match all antecedent atoms to the assertions stored in the `Extension Manager`, and for each match it derives one consequent atom. The `Merging Manager` component handles merging due to equalities of the form $s \approx t$ and $s \approx t@^{x}_{\leq n\, p.C}$, and it also implements the required pruning operation [27]. Furthermore, the `NI Manager` component examines annotations in equalities of the form $s \approx t@^{x}_{\leq n\, p.C}$ and determines whether s or t should be replaced with a root node due to the NI-rule; if so, it delegates the replacement process to the `Merging Manager` component.

The `Expansion Manager` component is responsible for applying the $\geq$-rule. This is done by invoking one of the (currently two) subcomponents that implement different strategies for dealing with existential quantifiers. The `Creation Order` expansion manager implements the standard way of dealing with existential quantifiers from tableau and hypertableau algorithms—that is, it introduces a new individual. In contrast, the `Individual Reuse` expansion manager tries to reduce the size of constructed pre-model (at the expense of additional nondeterminism) by reusing existing individuals [22]; we discuss this optimisation in more detail in Section 4.1. Both expansion managers can be parameterised with a blocking strat-

egy, allowing them to use various optimised blocking techniques as appropriate for the language that the ontology is expressed in. Supported blocking techniques include pairwise or single *anywhere blocking* [27] (see Section 4.2), *core blocking* [6] (see Section 4.2.2), and blocking via signature caching (see Section 4.2.1).

The `Datatype Manager` component checks the consistency of datatype constraints. These are represented as a set of assertions of the form $dt(s)$, $\neg dt(s)$, and $s_1 \not\approx s_2$, where s, s_1 and s_2 are *concrete individuals* (i.e., placeholders for data values), and dt is an explicit *enumeration* of data values $\{v_1, \ldots, v_n\}$ or an expression of the form $d[\varphi]$ for d a datatype (including the special datatype *rdfs:Literal*) and φ a *facet expression*. For example, assertion

$$\begin{array}{rl} \textit{xsd:integer}[\ \textit{xsd:minInclusive} & \text{“13”}\hat{}\hat{}\textit{xsd:integer} \\ \textit{xsd:maxExclusive} & \text{“15”}\hat{}\hat{}\textit{xsd:integer}\ ](s) \end{array}$$

uses the *xsd:minInclusive* and *xsd:maxExclusive* facets to restrict to integers 13 and 14 the data value that can be assigned to s. Given a set of such assertions, the `Datatype Manager` component uses the algorithm described in [23] to check whether each concrete individual can be assigned a data value in a way that satisfies all of the given assertions. An important aspect of this algorithm is modularity: one can easily add a new datatype without having to change the implementation of the existing datatypes.

The `DGraph Manager` component manages the information relevant to the description graph [21] extension to OWL 2. We describe description graphs and how they are handled in HermiT in more detail in Section 5.2.

The `Tableau` component orchestrates the pre-model construction process by delegating various subtaks to the relevant components. It first calls the `DGraph Manager` and the `Resolution Manager` components to derive all possible fresh facts. Each equality assertion derived during this step is immediately passed to the `Merging Manager` component, unless the equality assertion is also relevant for the NI-rule, in which case the equality assertion is buffered in the `NI Manager` component. All contradictions are determined eagerly (i.e., immediately after conflicting assertions are derived), so that backtracking can be initiated as soon as possible. Once all the DL-clauses have been applied, the `Tableau` component calls the `Datatype Manager` component to check consistency of datatype constraints; if that does not reveal a contradiction, the `Tableau` component then calls the `NI Manager` component to process all buffered annotated equalities. If any of these steps derives new facts, the entire process is repeated until a fixpoint is reached. Only after no new facts can be derived in this way, the `Tableau` component calls the `Expansion Manager` component in order to process assertions involving existential class expressions. If the latter step introduces new assertions, then the entire process is repeated; otherwise, reasoning terminates, at which point the `Extension Manager` contains a pre-model proving the consistency of $\mathcal{C} \cup \mathcal{A}$.

4 Optimising Ontology Consistency Tests

HermiT uses several optimisations that improve the efficiency of ontology consistency checking in typical cases. Of particular interest are *individual reuse* and several *blocking* optimisations, which we describe in more detail in this section.

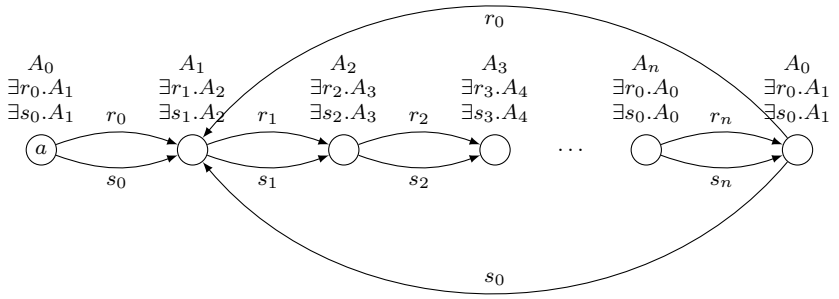


Fig. 3: A Pre-Model of $\mathcal{O}_1$ of Polynomial Size

HermiT also uses dependency-directed backtracking [1, Chapter 9], but this optimisation is standard so we do not discuss it here in more detail.

4.1 Individual Reuse

Individual reuse [22] is a technique whose goal is to reduce the size of the constructed pre-models at the expense of additional nondeterminism. Consider again the ontology $\mathcal{O}_1$ from Section 3.2. The the part of the ABox from Figure 2 above the dashed line is the canonical pre-model for $\mathcal{O}_1$, but it is exponential in the size of $\mathcal{O}_1$ and thus expensive to construct. However, the ABox shown in Figure 3 is polynomial in the size of $\mathcal{O}_1$, and it also satisfies all axioms of $\mathcal{O}_1$. More specifically, to satisfy assertions $\exists r_0.A_1(a)$ and $\exists s_0.A_1(a)$, we need assertions $r_0(a,b)$, $A_1(b)$, $s_0(a,c)$, and $A_1(c)$, but we can obtain a pre-model even if $b = c$—that is, if we *reuse* b when trying to satisfy $\exists s_0.A_1(a)$. This is possible because the axioms of $\mathcal{O}_1$ do not place many constraints on the shape of the pre-model.

Individual reuse exploits this idea in a systematic way. In particular, we associate with each class C a distinct individual t_C; moreover, we modify the $\geq$-rule so that satisfies each assertion of the form $\exists r.C(s)$ through assertions $r(s, t_C)$ and $C(t_C)$. Such a modification, however, can cause problems. For example, let $\mathcal{O}_3$ be the ontology obtained by extending the ontology $\mathcal{O}_1$ from Section 3.2 with axiom (13), which corresponds to DL-clause (14).

$$\exists r_0^-.\top \sqcap \exists s_0^-.\top \sqsubseteq \bot \tag{13}$$

$$r_0(y_1, x) \wedge s_0(y_2, x) \rightarrow \bot \tag{14}$$

It should be clear that $\mathcal{O}_3$ is consistent: the pre-model shown in Figure 2 satisfies $\mathcal{O}_3$ as well. The pre-model shown in Figure 3, however, does not satisfy $\mathcal{O}_3$ since DL-clause (14) is not satisfied when x is mapped to the individual labelled with A_1. In other words, the modified $\geq$-rule can overconstrain the pre-model, which makes the calculus unsound. In order to regain soundness, we make the modified $\geq$-rule nondeterministic: if reusing an individual leads to a clash, then we backtrack and introduce fresh individuals as usual. In our example, as soon as we detect that DL-clause (14) is not satisfied, we backtrack and introduce two fresh successors for a,

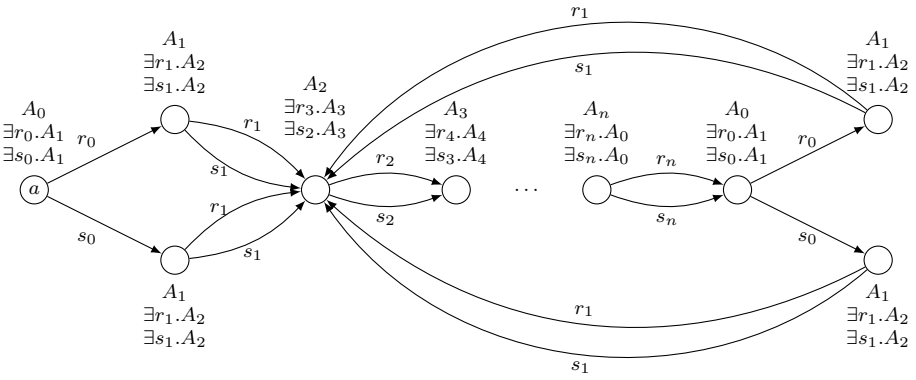


Fig. 4: A Pre-Model of $\mathcal{O}_3$ of Polynomial Size

but we still try to satisfy further existential restrictions by reusing individuals. In this way, we obtain the polynomially-sized pre-model shown in Figure 4.

Individual reuse can thus incur a significant degree of nondeterminism, but it can also considerably reduce the size of the constructed pre-models. As the results of our evaluation in Section 6 show, this technique is very effective on a range of ontologies. However, on ontologies that tightly constrain the shape of pre-models, the large number of nondeterministic choices causes considerable backtracking, which can render the technique impractical. We observed that this often happens on ontologies containing functional and inverse-functional properties. For example, if we extend $\mathcal{O}_3$ to make properties r_1 and s_1 inverse-functional, then all individuals labelled with A_1 must be merged, so the pre-model becomes as shown in Figure 3; then, DL-clause (14) is not satisfied, which triggers further backtracking.

4.2 Anywhere Blocking

As we discussed in Section 3.1, to use blocking we must ensure that there are no cyclic blocks; otherwise, it is not possible to 'copy-and-paste' the blocker's subtree. Prior to HermiT, this requirement was satisfied using *ancestor blocking* [15]: an individual s' could directly block an individual s only if, in addition to usual requirements on single or pairwise blocking, it is the case that s' is an ancestor of s. This, however, can lead to the generation of very large pre-models. For example, Figure 2 shows the pre-model for $\mathcal{O}_1$ obtained using single ancestor blocking, and, as one can see, the pre-model is exponential in the size of $\mathcal{O}_1$.

In order to reduce the size of pre-models constructed on typical ontologies, the hypertableau calculus uses *anywhere blocking* [27]. To this end, we assume that all individuals in an ABox are ordered according to some strict ordering $\prec$ compatible with the ancestor relation (i.e., we have $u \prec v$ whenever u is an ancestor of v). In practice, $\prec$ corresponds to the order in which individuals were introduced during derivation. Then, an individual s' can directly block an individual s if, in addition

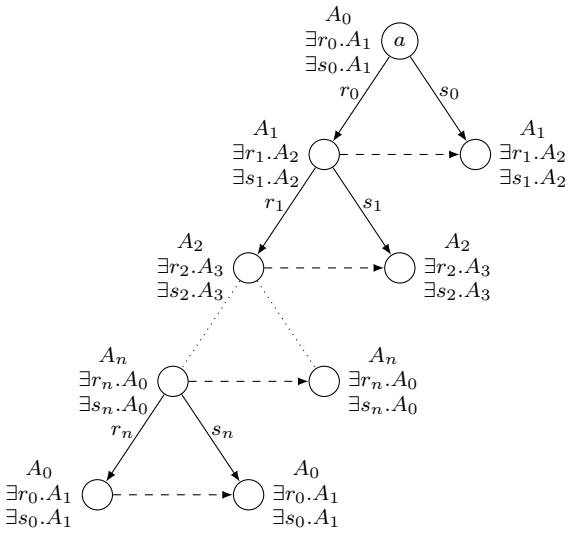


Fig. 5: A Pre-Model of $\mathcal{O}_1$ Obtained by Anywhere Single Blocking

to usual requirements on single or pairwise blocking, we also have that $s' \prec s$. Individuals s' and s thus do not need to be related through the ancestor relation, but absence of cyclic blocks is guaranteed since $\prec$ is acyclic.

In practice, ancestor blocking can considerably reduce the size of the constructed pre-models. For example, given the ontology $\mathcal{O}_1$ from Section 3.2, anywhere single blocking produces the pre-model shown in Figure 5, where dashed arrows point from blockers to blocked individuals. As one can see, switching from ancestor to anywhere blocking can reduce the pre-model size by an exponential factor. Moreover, the pre-models constructed using anywhere blocking are never larger than those constructed using ancestor blocking, and are often considerably smaller, so anywhere blocking is an important optimisation that allows HermiT to process many nontrivial ontologies.

4.2.1 Blocking via Signature Caching

Anywhere blocking enables a very effective caching technique that can considerably reduce the number of inferences that the reasoner performs over the course of its lifetime. We explain this technique by means of the example ontology $\mathcal{O}_1$ from Section 3.2, where we assume that we must check the satisfiability of each class A_i with $1 \leq i \leq n$. We can do this by iteratively checking the satisfiability of DL-clauses (9)–(10) and the assertion $A_i(a_i)$, where individuals a_i are fresh. Furthermore, let us assume that we initially determine that A_0 is satisfiable by constructing a pre-model $\mathcal{A}^0$ for (9)–(10) and $A_0(a_0)$. Finally, note that we can decide the satisfiability of A_1 by trying to construct a pre-model for $\mathcal{A}^0 \cup \{A_1(a_1)\}$: ontology $\mathcal{O}_1$ does contain nominals, so the assertions from $\mathcal{A}^0$ and the assertions obtained from $A_1(a_1)$ cannot interact. In other words, since $\mathcal{O}_1$ does not contain nominals, it enjoys the *disjoint model union* property: if I and J are two mod-

els of $\mathcal{O}_1$ and the domains of I and J are disjoint, then the union of I and J is still a model of $\mathcal{O}_1$. Starting from $\mathcal{A}^0 \cup \{A_1(a_1)\}$, however, has a distinct benefit: we can use individuals from $\mathcal{A}^0$ to block individuals in assertions obtained from $A_1(a_1)$, which can considerably speed up pre-model construction. In our example, the Hyp-rule will introduce assertions $\exists r_1.A_2(a_1)$ and $\exists s_1.A_2(a_1)$, which will be expanded into $r_1(a_1, u)$, $A_2(u)$, $r_2(a_1, v)$, and $A_2(v)$, and then individuals u and v will become blocked by individuals in $\mathcal{A}^0$; hence, we can decide the satisfiability of A_1 without recreating the full subtree under individual a_1.

HermiT implements a slightly more refined version of this idea. In particular, note that we do not really care about the assertions in pre-model $\mathcal{A}^0$; instead, we can just memorise the *signatures* of potential blockers. For single blocking, a signature is just a set of classes, and each individual s in an ABox $\mathcal{A}$ is associated with signature $\mathcal{L}_\mathcal{A}(s)$. For pairwise blocking, a signature is a 4-tuple of two sets of classes and two sets of object properties, and each individual s with predecessor t in an ABox $\mathcal{A}$ is associated with signature $\langle \mathcal{L}_\mathcal{A}(s), \mathcal{L}_\mathcal{A}(t), \mathcal{L}_\mathcal{A}(s,t), \mathcal{L}_\mathcal{A}(t,s)\rangle$. Blocking via signature caching now works as follows. Each time the hypertableau calculus produces a pre-model $\mathcal{A}$, for each non-blocked individual s in $\mathcal{A}$, we extract from $\mathcal{A}$ the signature for s and add it to a global cache. In each subsequent run of the hypertableau algorithm, if we determine that the global cache contains the signature of an individual u, then we know that u is blocked by some individual s from a previous run of the algorithm; otherwise, we determine the blocking status of u as usual. The global cache can be implemented easily using a hash table. This optimisation greatly improves performance of reasoning tasks that involve more than one run of the hypertableau algorithm on ontologies without nominals.

4.2.2 Core Blocking

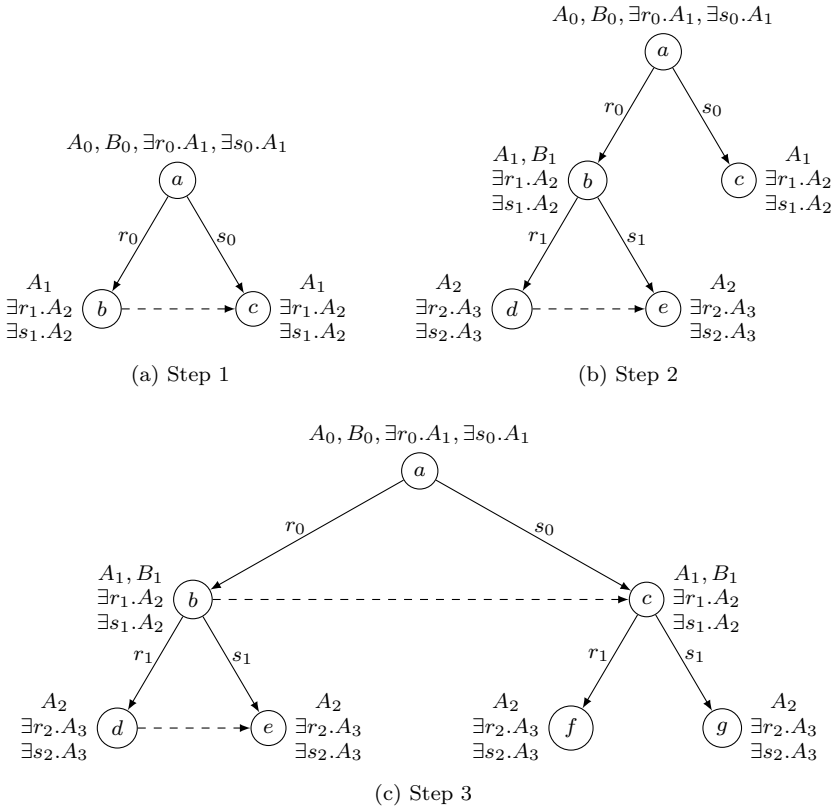
Core blocking [6] is another technique that HermiT uses in order to curb the size of the generated pre-models. It is based on an observation that, even if individuals s and s' do not satisfy the relevant direct blocking conditions from Section 3 exactly, it might still be possible to 'cut-and-paste' the subtree under s'. We explain this using an ontology $\mathcal{O}_4$ that is obtained by extending ontology $\mathcal{O}_1$ from Section 3.2 with axioms (15)–(16) instantiated for each $0 \leq i \leq n$.

$$\exists r_i.A_{(i+1) \bmod (n+1)} \sqsubseteq B_i \tag{15}$$
$$\exists s_i.A_{(i+1) \bmod (n+1)} \sqsubseteq B_i \tag{16}$$

These axioms correspond to DL-clauses (17)–(18) for $0 \leq i \leq n$.

$$r_i(x, y) \wedge A_{(i+1) \bmod (n+1)}(y) \rightarrow B_i(x) \tag{17}$$
$$s_i(x, y) \wedge A_{(i+1) \bmod (n+1)}(y) \rightarrow B_i(x) \tag{18}$$

We next discuss the construction of a derivation for $\mathcal{O}_4$. After applying the hypertableau rules to assertions involving individual a, we obtain the ABox shown in Figure 6a. At this point, the labels of individuals b and c coincide, so individual b blocks individual c. Next, we apply the $\geq$-rule to b, and then we apply the Hyp-rule as long as possible; this produces the ABox shown in Figure 6b. Labels of individuals b and c do not coincide any more, so individual c is not blocked by individual b, and so the $\geq$-rule must be applied to c, and by exhaustively applying



(a) Step 1

(b) Step 2

(c) Step 3

Fig. 6: Core Blocking Example

the Hyp-rule to c we derive $B_1(c)$, as shown in Figure 6c. The derivation of $B_1(c)$, however, depends on the successors of c, and it does not lead to the derivation of other assertions involving an ancestor of c; therefore, already in the ABox shown in Figure 6a, we can 'cut' the subtree under b and 'paste' it under c since doing so does not enable any additional hypertableau inferences. In other words, we can let b block c in the ABox shown in Figure 6b even though the labels of the two individuals do not coincide exactly.

More generally, an individual s' can block an individual s if no hypertableau inferences are enabled when we temporarily replace the subtree under s with the direct successors of s'. A naïve way to apply this idea is to consider each pair of individuals s and s', for each pair conduct the temporary replacement, and then check whether any of the DL-clauses are applicable to s; if not, then we declare that s is blocked by s'. Such an approach, however, would be very inefficient since it would consider a quadratic number of individual pairs.

To obtain a practical solution, we use a heuristic to identify pairs of individuals that are likely to block each other; we call these *candidate* pairs. Towards this goal, each assertion in an ABox is assigned a flag specifying whether the assertion is *core* or not. There are several ways to determine the value of this flag, all of which are

discussed in more detail in [6]; intuitively, all of these approaches aim to identify the assertions that determine the 'properties' of the involved individuals, with the expectation that similar individuals should occur in similar core assertions. In our example, concepts A_i would belong to the core, whereas concepts B_i would not. Then, in order to determine candidate pairs of individuals, we compare the core assertions of the two individuals. For each thus obtained candidate pair of s and s', we then conduct the check from the previous paragraph and determine with certainty whether s' can block s.

5 Supporting Additional Features

We next discuss several features supported in HermiT that go beyond the OWL 2 DL standard, but that have been identified as very useful in practice.

5.1 SWRL Rules

The Semantic Web Rule Language (SWRL) extends OWL with rules, which can capture non-tree-like axioms [26] such as 'all uncles of a person share the same father, who is also that person's grandfather' [14]. Although reasoning in OWL ontologies extended with SWRL rules is undecidable in general, restricting the application of rules to individuals explicitly named in the input ABox—that is, treating the rules as being *DL-safe*—ensures decidability [26].

HermiT's reasoning algorithm is based on DL-clauses, which are essentially rules, so extending the reasoning algorithm to handle SWRL rules requires minimal effort. The main challenge is to respect the DL-safety requirement, which is achieved by extending each rule r with an atom $O(x)$ for each variable x occurring in r, where O is a internal predicate used by HermiT to qualify all individuals occurring in the input ABox. After this modification, the Hyp-rule can be used as usual to apply the rules and derive their consequences.

A problem can arise, however, if SWRL rules use transitive properties, or properties defined via property chain axioms, and in these circumstances HermiT is, in general, incomplete. An extension of the automata-based encoding of properties has recently been developed that solves this problem for transitive properties [4], and we conjecture that this approach can be extended to correctly handle property chains as well; however, the approach has not yet been implemented in HermiT.

5.2 Description Graphs

Description Graphs are an extension of OWL designed facilitate more precise modelling of arbitrarily connected structures [21]. Reasoning with an unrestricted extension of OWL 2 DL with description graphs is undecidable, but several decidable restrictions of the general formalism have been developed [21]. Roughly speaking, these ensure that ontologies extended with description graphs can axiomatise structures of unbounded size, but whose non-tree-like components are all bounded in size; the latter observation can then be used to define a suitable notion of blocking and thus ensure termination of reasoning.

Description graphs have also been implemented in HermiT. Given an OWL 2 DL ontology extended with description graphs and rules, HermiT can determine whether the restrictions necessary for decidability are satisfied; if so, the ontology can be used in all standard reasoning tasks. Since there is no standard syntax for description graphs, these can at present be passed to HermiT only via a Java API.

5.3 SPARQL Queries

The SPARQL query language [30] and its revision SPARQL 1.1 [10] provide a standard query interface for Semantic Web systems. The SPARQL 1.1 version of the language includes several *entailment regimes* [9,8] that support querying implicit information logically entailed by a system's ontology and the data. The OWL 2 Direct Semantics Entailment Regime has been specifically designed to enable querying OWL 2 DL ontologies interpreted under the Direct Semantics [24] of OWL 2 DL, and is thus of particular interest for OWL reasoners.

HermiT supports SPARQL queries via the OWL-BGP SPARQL wrapper[2]—a separate library that implements SPARQL query answering using HermiT's public interface [18]. The wrapper is based on the Apache Jena SPARQL processor ARQ,[3] and it can be used with any reasoner that supports the OWL API; however, HermiT also provides several methods for retrieving ontology statistics that the wrapper uses to produce near-optimal query evaluation plans [18].

6 Evaluation

We compared HermiT 1.3.7 with the state of the art tableau reasoners Pellet 2.3.0 [33] and FaCT++ 1.6.1 [35]. Pellet and FaCT++ are based on tableau algorithms [15], so they use a different set of derivation rules and (possibly) a different blocking strategy than HermiT. The purpose of our evaluation was mainly to compare the behaviour of the hypertableau calculus and our reasoning optimisations with that of the tableau calculi and other optimisations in the other systems, and furthermore to demonstrate the advantages of HermiT on a certain set of ontologies. Individual reuse is not enabled by default because its performance can be unpredictable; however, on some ontologies it offers considerable performance improvements, so we also tested a version called HermiT-IR in which individual reuse is always switched on. Our tests involved primarily class classification—a core reasoning service for all reasoners that provides us with a natural measure of a reasoner's performance—but we also measured the time needed to check the consistency of all classes in several ontologies, as this allowed us to analyse the impact of consistency test optimisations independently from the impact of classification optimisations. We did not measure property classification times since, to the best of our knowledge, the other reasoners are complete on that task [7].

We used a repository of standard ontologies, mainly from the Open Biological Ontologies (OBO) Foundry,[4] the Gardiner ontology suite [5], the Phenoscape

[2] `http://code.google.com/p/owl-bgp/`

[3] `http://jena.apache.org/documentation/query/index.html`

[4] `http://obofoundry.org/`

Table 1: Statistics of some interesting ontologies

ID	Name	DL	C	P	T	R
00001	ACGT-v1.0	$\mathcal{SROIQ}(D)$	1,751	265	5,329	128
00004	BAMS-simplified	$\mathcal{SHIF}$	1,110	12	18,813	9
00024	DOLCE	$\mathcal{SHOIN}(D)$	209	317	1,210	334
00026	GALEN-no-FIT	$\mathcal{ELH}$	23,141	949	35,531	958
00029	GALEN-doctored	$\mathcal{ALEHIF}^+$	2,748	413	4,320	442
00032	GALEN-undoctored	$\mathcal{ALEHIF}^+$	2,748	413	4,563	442
00285	FMA-constitutional	$\mathcal{ALCOIF}(D)$	41,648	168	123,090	0
00347	LUBM-one-uni	$\mathcal{ALEHI}^+(D)$	43	32	88	6
00350	OBI	$\mathcal{SHOIN}(D)$	2,638	83	9,876	50
00351	AERO	$\mathcal{SROIQ}(D)$	276	66	460	45
00354	NIF-gross-anatomy	$\mathcal{SROIF}(D)$	4,042	77	6,581	49
00463	Fly-anatomy-XP	$\mathcal{SRI}$	8,023	27	16,020	23
00471	FMA-lite	$\mathcal{EL}++$	78,983	8	121,709	4
00477	Gazetteer	$\mathcal{EL}++$	150,981	5	167,351	2
00512	Lipid	$\mathcal{ALCHIN}$	716	46	2,349	26
00545	Molecule-role	$\mathcal{EL}++$	9,223	3	9,629	2
00774	RNA-v0.2	$\mathcal{SRIQ}(D)$	244	93	581	102
00775	Roberts-family	$\mathcal{SROIQ}(D)$	61	87	239	85
00778	SNOMED	$\mathcal{SH}$	54,974	9	54,974	4
00786	NCI-v12.04e	$\mathcal{SH}(D)$	93,413	206	130,928	19

Project,[5] and several variants of the GALEN ontology [31]. We preprocessed all ontologies to resolve ontology imports so that each test ontology is contained in a single file loadable though the OWL API. Each test ontology is assigned a unique ontology ID, which can be used to download the ontology from our ontology repository.[6] Please note that each ontology ID identifies a particular self-contained and 'frozen' OWL ontology file—that is, a different version of the same ontology, or a version that imports different ontologies, is assigned a different ontology ID.

We did not consider ontologies that contain datatypes outside the OWL 2 datatype map, ontologies that are inconsistent, or ontologies that are too simple to provide any useful test of performance (i.e., that contain few axioms and/or classes). We thus selected 484 consistent OWL 2 ontologies from our repository, with expressivity ranging from DL-Lite and $\mathcal{EL}$ to $\mathcal{SROIQ}(D)$, and containing between 100 and 2,492,761 axioms, between 40 and 244,232 classes, and between 1 and 2,259 properties. Several commonly-used and well-known ontologies from our test suite are shown in Table 1, with the table columns showing the ontology ID, a human-readable name (Name), the DL expressivity (DL), the number of classes (C), the number of properties (P), the number of TBox axioms (T), and the number of RBox axioms (R).

All experiments were run on a Dell T7600 workstation with two quad core Intel Xeon processors running at 3.30GHz under 64bit Linux. We used Java 1.6 with 12 GB of heap memory per test. Each test was allowed at most 30 minutes to complete. For each test, we loaded the ontology using the OWL API, passed it to the reasoner, and then measured the wall-clock time for each test. Most reasoners

[5] `http://www.phenoscape.org/`

[6] Ontology with ID 'xxxxx' is avalable at `http://www.cs.ox.ac.uk/isg/ontologies/UID/xxxxx.owl`.

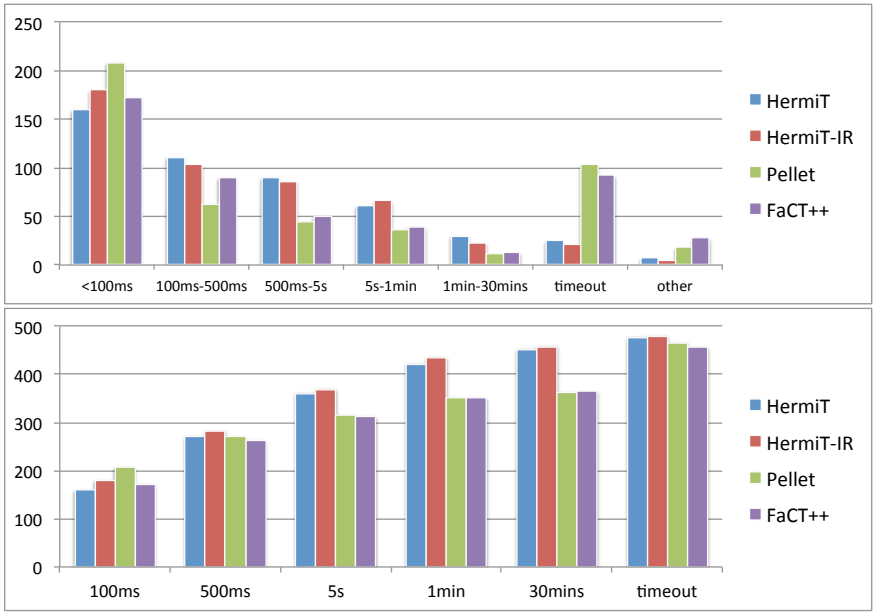


Fig. 7: Numbers of classified ontologies grouped by reasoning times

pre-process ontologies as they read them from the OWL API, and this can be seen as a form of reasoning, so we included this time into each test. Finally, classification times may vary considerably from run to run with all of the reasoners considered, so all of the presented results were obtained as an average over two runs.

Figure 7 shows the results of the ontology classification tests. The upper chart shows the number of ontologies successfully classified by a reasoner grouped by reasoning times shown on the horizontal axis. The number of ontologies that each reasoner failed to classify either due to memory exhaustion or an internal error are shown in the 'other' group. The lower chart shows the cumulative numbers of successfully classified ontologies grouped by the reasoning time. Please note that the scales of the axes in the two charts are different.

Figure 7 shows that the ontologies in our repository are quite diverse: many can be classified within 100ms, but quite a few turned out to be 'hard' for the reasoners. There are 20 ontologies that no reasoner could classify with the given resource bounds; these include the GALEN ontology and several of its modules, two extensions of the GO ontology, the OMEO ontologies, and several Phenoscape ontologies. Pellet and FaCT++ were able to classify more ontologies than HermiT in 100ms. Even without individual reuse, however, HermiT was able to classify 64 ontologies that the other two reasoners were unable to process within the given time and memory bounds; these include several OBO ontologies, a large number of Phenoscape ontologies, and the majority of the extensions to the GO ontology. Table 2 shows for each major group of ontologies in our repository the number of ontologies successfully classified by HermiT (H), HermiT with individual reuse (H-IR), Pellet (P), and FaCT++ (F).

Table 2: Numbers of classified ontologies by major ontology groups

Group	ID	total	H	H-IR	P	F
DOLCE	00013–00024	9	7	9	0	9
corollary	00026–00039	14	5	5	2	5
GO extensions	00040–00048	9	7	7	0	0
Gardiner corpus	00049–00346	68	67	68	64	66
OBO	00351–00697	290	286	287	266	244
Phenoscape	00700–00771	66	58	58	10	16

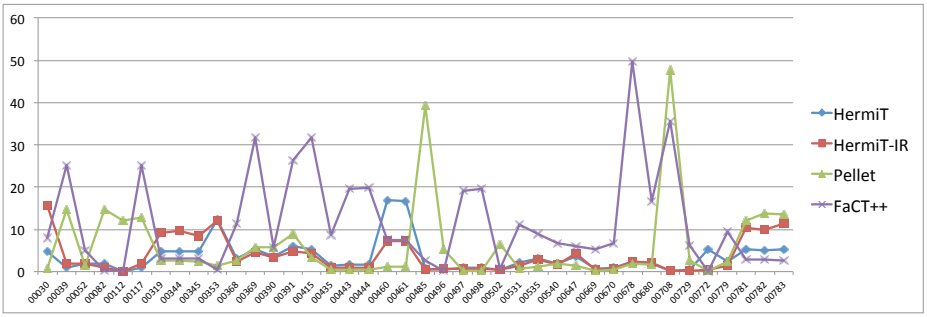


Fig. 8: Classification times of easy ontologies in seconds.

Figure 8 shows the classification times for ontologies on which all reasoners took less than one minute, and at least one reasoner took more than five seconds; the horizontal axis shows the ontology IDs. As one can see, the performance of HermiT is comparable to that of the other reasoners.

We next focus on the 'hard' ontologies—that is, ontologies that at least one reasoner took more than a minute to classify. For brevity and clarity, we present the results only for 20 common ontologies in this group. For some 'hard' ontologies (e.g., for OBO ontologies), our repository contains several different versions of the same ontology, in which case we report the results only for the newest available version. Table 3 summarises our results. We identified four groups of these ontologies, which we delineate in Table 3 using horizontal lines.

The first group contains ontologies on which HermiT-IR is much faster than HermiT. There are four ontologies that HermiT failed to classify but HermiT-IR processed successfully, which demonstrates the practical benefits of individual reuse. Furthermore, individual reuse allows HermiT to classify BAMS-simplified (ID 00004), NIF-gross-anatomy (ID 00354), and Fly-anatomy-XP (ID 00463) much faster than the other two reasoners; however, FaCT++ outperforms HermiT-IR on ACGT-v1.0 (ID 00001), DOLCE (ID 00024), and OBI (ID 00350). All of these ontologies contain nominals, which prevents HermiT from caching blocking labels.

The second group contains ontologies on which HermiT (without individual reuse) outperforms the other two reasoners. The good performance of HermiT in these cases is largely due to the caching of blocking labels: only the first test is hard, and all subsequent tests are easy because early blocking (facilitated through caching) prevents the creation of large pre-models. With individual reuse, however, this optimisation is ineffective, as most individuals in the model are root

Table 3: Results of the Class Classification Tests

ID	Classification Times (seconds)			
	HermiT	HermiT-IR	Pellet	FaCT++
00001	31.9	14.7	100.5	1.3
00004	296.7	0.3	timeout	147.0
00024	timeout	5.1	timeout	0.7
00285	timeout	543.4	timeout	exception
00350	209.5	47.9	timeout	6.6
00354	timeout	109.4	timeout	1,571.6
00463	timeout	24.8	timeout	240.7
00477	56.4	18.5	30.8	timeout
00029	1.7	3.0	timeout	3.3
00032	5.3	67.6	out of mem.	11.2
00786	69.8	753.3	372.1	100.1
00347	1.2	1.2	2.5	191.3
00351	0.4	0.3	timeout	0.2
00471	18.2	19.1	77.1	35.9
00545	1.8	1.4	0.7	168.7
00774	0.7	0.7	78.3	0.3
00026	timeout	timeout	21.8	timeout
00512	timeout	timeout	0.4	1.1
00775	out of mem.	out of mem.	timeout	42.6
00778	timeout	timeout	out of mem.	1,049.3

individuals and cannot be used as blockers. Each subsumption test with individual reuse takes less time than the 'hard' test without individual reuse; however, the cumulative slowdown due to the lack of caching is detrimental.

The third group contains ontologies on which HermiT-IR performs similarly to HermiT, thus suggesting little impact of individual reuse. The performance of HermiT on these ontologies, if not always the best, is competitive with the other reasoners. To distinguish the impact of consistency test optimisations from the impact of classification optimisations, for each of these ontologies we measured the time needed to iteratively determine the consistency of all classes in the ontologies, which can benefit only from consistency test optimisations; Table 4 shows the results of these tests. On LUBM-one-uni (ID 00347) and Molecule-role (ID 00545), HermiT determined class consistency in time similar to FaCT++, whereas HermiT outperformed FaCT++ on these ontologies during classification; we believe that the latter is mainly due to our classification optimisations. Furthermore, we believe that the same observation explains why HermiT classified FMA-lite (ID 00471) and RNA-v0.2 (ID 00774) faster than Pellet. Finally, FaCT++ determined the consistency of all classes one or two orders of magnitude quicker than HermiT on AERO (ID 00351), FMA-lite, and RNA-v0.2. However, HermiT is competitive with FaCT++ on these ontologies, which we also believe to be mainly due to our optimised classification algorithm.

The fourth group contains ontologies that HermiT failed to classify, even with individual reuse. In contrast, Pellet and/or FaCT++ could classify some of these ontologies, sometimes easily. In particular, HermiT failed to classify the Lipid (ID 00512) ontology, which poses no problems for Pellet or FaCT++. Our analysis revealed that the main bottleneck in HermiT on this ontology is due to a large number of individual merges encountered during the pre-model construction—an effect that does not seem to occur in other test ontologies.

Table 4: Results of the Consistency Tests

ID	Times for the Consistency Test (seconds)			
	HermiT	HermiT-IR	Pellet	FaCT++
00347	1.0	1.0	1.2	1.1
00351	0.5	0.3	timeout	0.009
00471	61.4	16.4	4.4	3.5
00545	2.0	1.6	0.7	1.8
00774	0.7	0.5	0.6	0.02

To summarise, although HermiT is not universally better than Pellet and FaCT++ on all test ontologies, our results indicate that HermiT may be more robust on 'hard' ontologies as it managed to successfully process more ontologies. Moreover, individual reuse seems to be particularly useful on such ontologies and can significantly improve HermiT's performance.

7 Conclusions

In this system description paper we have presented the HermiT OWL 2 reasoner and its architecture, and have briefly described several novel optimisation techniques used to improve HermiT's performance; these include optimisations of single consistency tests as well as optimisations of complex reasoning tasks. Finally, we discussed HermiT's support for several features that go beyond OWL 2, such as SWRL rules, description graphs, and SPARQL queries. Hence, this paper provides a comprehensive overview of the system, which will hopefully help users and developers to better understand the various options and extension points available.

We have also presented the results of a detailed performance comparison of of HermiT, FaCT++, and Pellet on a wide range of ontologies. All test ontologies used are accessible via immutable URIs, thus allowing our tests to be easily repeated. Although HermiT does not outperform the other reasoners on all ontologies, its performance seems to be more consistent, particularly on 'hard' ontologies.

Up-to-date information can be found on the Hermit's website,[7] and its Google code project[8] provides an issue tracker and a discussion forum.

Acknowledgements This work was supported by the EPSRC projects ExODA and MaSI[3].

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook, 2nd edn. Cambridge University Press (2007)
2. Baader, F., Sattler, U.: An Overview of Tableau Algorithms for Description Logics. Studia Logica **69**, 5–40 (2001)
3. Cuenca Grau, B., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P.F., Sattler, U.: OWL 2: The next step for OWL. Journal of Web Semantics **6**(4), 309–322 (2008)
4. Cuenca Grau, B., Motik, B., Stoilos, G., Horrocks, I.: Computing Datalog Rewritings Beyond Horn Ontologies. In: F. Rossi (ed.) Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI 2013), pp. 832–838. Beijing, China (2013)

[7] `http://www.hermit-reasoner.com/`

[8] `http://code.google.com/p/hermit-reasoner/`

5. Gardiner, T., Horrocks, I., Tsarkov, D.: Automated Benchmarking of Description Logic Reasoners. In: Proc. of the 2006 Int. Workshop on Description Logic (DL 2006), *CEUR Workshop Proceedings*, vol. 189 (2006)
6. Glimm, B., Horrocks, I., Motik, B.: Optimized Description Logic Reasoning via Core Blocking. In: J. Giesl, R. Hähnle (eds.) Proc. of the 5th Int. Joint Conf. on Automated Reasoning (IJCAR 2010), *Lecture Notes in Computer Science*, vol. 6173, pp. 457–471. Springer, Edinburgh, UK (2010)
7. Glimm, B., Horrocks, I., Motik, B., Shearer, R., Stoilos, G.: A Novel Approach to Ontology Classification. Journal of Web Semantics **14**, 84–101 (2012)
8. Glimm, B., Krötzsch, M.: SPARQL Beyond Subgraph Matching. In: Proc. of the 9th Int. Semantic Web Conf. (ISWC 2010), *Lecture Notes in Computer Science*, vol. 6414, pp. 241–256. Springer (2010)
9. Glimm, B., Ogbuji, C.: SPARQL 1.1 Entailment Regimes. W3C Recommendation (2013). Available at `http://www.w3.org/TR/sparql11-entailment/`
10. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (2013). Available at `http://www.w3.org/TR/sparql11-query/`
11. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. Chapman & Hall/CRC (2009)
12. Horridge, M., Bechhofer, S.: The OWL API: A Java API for OWL ontologies. Semantic Web Journal **2**(1), 11–21 (2011)
13. Horrocks, I., Patel-Schneider, P.F.: Reducing OWL entailment to description logic satisfiability. Journal of Web Semantics **1**(4), 345–357 (2004)
14. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission (2004). Available at `http://www.w3.org/Submission/SWRL/`
15. Horrocks, I., Sattler, U.: A Tableau Decision Procedure for $\mathcal{SHOIQ}$. Journal of Automated Reasoning **39**(3), 249–276 (2007)
16. Hudek, A.K., Weddell, G.E.: Binary absorption in tableaux-based reasoning for description logics. In: B. Parsia, U. Sattler, D. Toman (eds.) Proc. of the 2006 Int. Workshop on Description Logics (DL 2006), *CEUR Workshop Proceedings*, vol. 189. Windermere, UK (2006)
17. Hustadt, U., Motik, B., Sattler, U.: Data Complexity of Reasoning in Very Expressive Description Logics. In: L.P. Kaelbling, A. Saffiotti (eds.) Proc. of the 19th Int. Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 466–471. Morgan Kaufmann Publishers, Edinburgh, UK (2005)
18. Kollia, I., Glimm, B.: Optimizing SPARQL Query Answering over OWL Ontologies. Journal of Artificial Intelligence Research **48**, 253–303 (2013)
19. Krötzsch, M., Simančík, F., Horrocks, I.: A Description Logic Primer. Computing Research Repository (CoRR) **abs/1201.4089** (2012)
20. Kutz, O., Horrocks, I., Sattler, U.: The Even More Irresistible $\mathcal{SROIQ}$. In: P. Doherty, J. Mylopoulos, C.A. Welty (eds.) Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006), pp. 68–78. AAAI Press, Lake District, UK (2006)
21. Motik, B., Cuenca Grau, B., Horrocks, I., Sattler, U.: Representing ontologies using description logics, description graphs, and rules. Artificial Intelligence Journal **173**(14), 1275–1309 (2009)
22. Motik, B., Horrocks, I.: Individual Reuse in Description Logic Reasoning. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2008), *Lecture Notes in Computer Science*, vol. 5195, pp. 242–258. Springer (2008)
23. Motik, B., Horrocks, I.: OWL Datatypes: Design and Implementation. In: Proc. of the 7th Int. Semantic Web Conference (ISWC 2008), *Lecture Notes in Computer Science*, vol. 5318, pp. 307–322. Springer (2008)
24. Motik, B., Patel-Schneider, P.F., Cuenca Grau, B.: OWL 2 Web Ontology Language Direct Semantics (Second Edition). W3C Recommendation (2012). Available at `http://www.w3.org/TR/owl2-direct-semantics/`
25. Motik, B., Patel-Schneider, P.F., Parsia, B. (eds.): OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation (2012). Available at `http://www.w3.org/TR/owl2-syntax/`
26. Motik, B., Sattler, U., Studer, R.: Query Answering for OWL-DL with Rules. Journal of Web Semantics **3**(1), 41–60 (2005)

27. Motik, B., Shearer, R., Horrocks, I.: Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research **36**, 165–228 (2009)
28. Nonnengart, A., Weidenbach, C.: Computing Small Clause Normal Forms. In: J.A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, vol. 1, chap. 6, pp. 335–367. Elsevier and MIT Press (2001)
29. Plaisted, D.A., Greenbaum, S.: A Structure-Preserving Clause Form Translation. Journal of Symbolic Computation **2**(3), 293–304 (1986)
30. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008). Available at `http://www.w3.org/TR/rdf-sparql-query/`
31. Rector, A., Gangemi, A., Galeazzi, E., Glowinski, A.J., Mori, A.R.: The GALEN CORE Model Schemata for Anatomy: Towards a Re-usable Application-Independent Model of Medical Concepts. In: Proc. of the 12th Int. Congress of the European Federation for Medical Informatics (MIE 1994), pp. 229–233 (1994)
32. Simančík, F.: Elimination of Complex RIAs without Automata. In: Y. Kazakov, D. Lembo, F. Wolter (eds.) Proc. of the 2012 Int. Workshop on Description Logics (DL 2012), *CEUR Workshop Proceedings*, vol. 846. Rome, Italy (2012)
33. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A Practical OWL-DL Reasoner. Journal of Web Semantics **5**(2), 51–53 (2007)
34. Tsarkov, D., Horrocks, I.: Efficient Reasoning with Range and Domain Constraints. In: V. Haarslev, R. Möller (eds.) Proc. of the 2004 Int. Workshop on Description Logics (DL 2004), *CEUR Workshop Proceedings*, vol. 104 (2004)
35. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Proc. of the 6th Int. Joint Conf. on Automated Reasoning (IJCAR 2006), *Lecture Notes in Artificial Intelligence*, vol. 4130, pp. 292–297. Springer (2006)
36. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimizing Terminological Reasoning for Expressive Description Logics. Journal of Automated Reasoning **39**(3), 277–316 (2007)