Pellet: A Practical OWL-DL Reasoner

Evren Sirin [a] , Bijan Parsia [a] , Bernardo Cuenca Grau [a,b] ,
Aditya Kalyanpur [a] , Yarden Katz [a]

[a] *University of Maryland, MIND Lab, 8400 Baltimore Ave, College Park MD 20742, USA*

[b] *Departamento de Informatica, Universidad de Valencia Av. Vicente Andres Estelles, s/n, 46100 Burjassot, Valencia, SPAIN*

Abstract

In this paper, we present Pellet: a complete and capable OWL-DL reasoner with acceptable to very good performance, extensive middleware, and a number of unique features. Pellet is written in Java and is open source under a very liberal license. It is used in a number of projects, from pure research to industrial settings.

Pellet is the first sound and complete OWL-DL reasoner with extensive support for reasoning with individuals (including nominal support and conjunctive query), user-defined datatypes, and debugging support for ontologies. It implements several extensions to OWL-DL including a combination formalism for OWL-DL ontologies, a non-monotonic operator, and preliminary support for OWL/Rule hybrid reasoning. It has proven to be a reliable tool for working with OWL-DL ontologies and experimenting with OWL extensions.

In this paper we describe Pellet's features, architecture and special capabilities, along with an empirical comparison of its performance against other leading OWL-DL reasoners.

Key words: Web Ontology Language, Description Logics Reasoning, Tableau-based Theorem Proving, Semantic Web

1 Introduction

OWL is a World Wide Web Consortium (W3C) Recommendation for representing ontologies on the Semantic Web. The OWL-DL sublanguage is a syntactic variant of the Description Logic $\mathcal{SHOIN}(\mathcal{D})$, that is, an OWL-DL ontology corresponds

Email addresses: evren@cs.umd.edu (Evren Sirin), bparsia@isr.umd.edu (Bijan Parsia), bernardo@mindlab.umd.edu (Bernardo Cuenca Grau), aditya@cs.umd.edu (Aditya Kalyanpur), yarden@umd.edu (Yarden Katz).

to a $\mathcal{SHOIN}(\mathcal{D})$ knowledge base. When OWL went to "Candidate Recommendation", there was concern within the working group that implementing a standard tableau based reasoner for OWL-DL would be too difficult for people not already experts in Description Logics or theorem proving. We began work on the OWL-DL reasoner, Pellet, to lay these concerns to rest. In a matter of months, we had a reasoner that passed a substantial number of the OWL test cases and even useful for reasoning with small, relatively simple ontologies. Over the next two years, Pellet has undergone continuous, if part time, development. Pellet was perhaps, as a very generous guess-estimate, 1.5 person years of effort by people who had no prior experience with Description Logics and little prior experience with theorem proving (especially not with tableau-based methods). This is not an unreasonable amount of effort for a production quality tool for a Recommendation of this complexity. [1]

Pellet is now a complete and capable OWL-DL reasoner with acceptable to very good performance, extensive middleware, and a number of unique features. It is written in Java and is open source under a very liberal license. It is used in a number of projects, from pure research to industrial settings. In this paper, we describe Pellet's features, architecture, and special capabilities, along with an empirical comparison of its performance against other leading OWL-DL reasoners.

Pellet is the first implementation of the full decision procedure for OWL-DL (including instances) and has extensive support for reasoning with individuals (including conjunctive query over assertions), user-defined datatypes, and debugging ontologies. It implements several extensions to OWL-DL including a combination formalism for OWL-DL ontologies, a non-monotonic operator, and preliminary support for OWL/Rule hybrid reasoning. It has proven to be a reliable tool for working with OWL-DL ontologies and experimenting with OWL extensions.

The rest of the paper is organized as follows: In Section 2, we discuss the basic services that can and should be provided by an OWL-DL reasoning component, followed, in Section 3, by a description of the architecture of Pellet. Sections 4 and 5 present, respectively, Pellet's support for ontology analysis and repair (especially, debugging) and Pellet's support for key extensions to the OWL-DL language. Section 6 covers the new optimizations introduced in Pellet, while Section 7 compares Pellet's performance with RacerPro and FaCT++.

2 Pellet as an OWL-DL Reasoner

The OWL Web Ontology Test Cases W3C Recommendation [1] defines two sorts of OWL "document checkers": OWL syntax checkers and OWL consistency checkers. It also defines four conformance classes of consistency checkers, OWL Lite/DL/Full

[1] The system can be downloaded from http://www.mindswap.org/2003/pellet/download.shtml

consistency checkers, and complete OWL-Lite consistency checkers. To be a consistency checker is to be sound with respect to the specific species' semantics. To be complete is to be a decision procedure with respect to that semantics. OWL-Full is not decidable, so there is no such thing as a complete OWL-Full consistency checker. It is a bit odd that the Test Cases Recommendation does not explicitly define a complete OWL-DL consistency checker, though perhaps explainable that, at the time, there was no known decision procedure for OWL-DL. Pellet is a complete OWL-DL consistency checker and a very incomplete OWL-Full consistency checker. It is also an OWL syntax checker. To our knowledge, Pellet is the first, and currently the only, complete OWL-DL consistency checker and has the most coverage of OWL as a whole of any reasoner (though some reasoners, particular OWL-Full ones, cover areas of OWL-Full reasoning Pellet just does not try to handle). Our implementation validates the design of the WebOnt working group.

Meeting the conformance criteria of a specification is laudable, but it does not necessarily result in a practical tool. The OWL Test Cases document [1] defines an OWL consistency checker as follows:

> An *OWL consistency checker* takes a document as input, and returns one word being Consistent, Inconsistent, or Unknown.

But, while consistency checking is an important task, it does not, in itself, allow one to *do* anything interesting with an ontology. Traditionally, in the ontology and Description Logic community, there is a suite of *inference services* held to be key to most applications or knowledge engineering efforts. Given that OWL-DL is a syntactic variant of the very expressive Description Logic $\mathcal{SHOIN}(\mathcal{D})$, it is imperative that a practical OWL reasoner provide at least the "standard" set of Description Logic inference services, namely:

- *Consistency checking*, which ensures that an ontology does not contain any contradictory facts. The OWL Abstract Syntax & Semantics document [2] provides a formal definition of ontology consistency that Pellet uses. In DL terminology (see Figure 1), this is the operation to check the consistency of an ABox with respect to a TBox. [2]
- *Concept satisfiability*, which checks if it is *possible* for a class to have any instances. If class is unsatisfiable, then defining an instance of the class will cause the whole ontology to be inconsistent.
- *Classification*, which computes the subclass relations between every named class to create the complete class hierarchy. The class hierarchy can be used to answer queries such as getting all or only the direct subclasses of a class.
- *Realization*, which finds the most specific classes that an individual belongs to; or in other words, computes the direct types for each of the individuals. Realization can only be performed after classification since direct types are defined with

[2] This corresponds to being an OWL consistency checker.

Abbr.	Stands for	Meaning
ABox	Assertional Box	Component that contains assertions about individuals, i.e. OWL facts such as type, property-value, equality or inequality assertions.
TBox	Terminological Box	Component that contains axioms about classes, i.e. OWL axioms such as subclass, equivalent class or disjointness axioms.
KB	Knowledge Base	A combination of an ABox and a TBox, i.e. a complete OWL ontology.

Fig. 1. Explanation of some commonly used terms in DL jargon

respect to a class hierarchy. Using the classification hierarchy, it is also possible to get all the types for that individual.

These services are inter-definable [3], but it is standard to reduce them all to consistency checking, as Pellet does. These basic services can be accessed by querying the reasoner. Generally, such queries are supported via an API such as the DIG interface [4]. Pellet supports the standard array of derivative queries and various reasoner management services both via its own API and by supplying bindings and support for common toolkits (such as Jena [5], WonderWeb OWL API [6] and DIG [4], see Section 3 for details).

Pellet also supports some less standard services. For example, while classification requires a degree of entailment support (i.e., certain subclass relations are *entailed* by the ontology and classification is the inferring of those relations), it generally is quite restricted. Only a very limited set of types of entailment are supported, though, in principle, arbitrary entailment between OWL documents can be reduced to the core service of consistency checking. In [7], the general entailment problem for OWL-DL is reduced to KB consistency problem by means of an appropriate transformation. Pellet has explicit support for testing arbitrary entailments using this technique.

Similarly, it is possible to reduce ABox conjunctive query answering to consistency checking. Queries about instances that are written in languages such as RDQL [8] or SPARQL [9] fall into this category. Since DLs have generally focused on reasoning with classes, queries about instances get much less emphasis in the literature and in implementations (though that is changing). As a consequence, there is not a lot of implementation experience or known optimization techniques in this area. In Pellet, we have implemented a somewhat optimized conjunctive query answering procedure.

We also have gone beyond both the standard set of inference services (consistency, satisfiability, classification, and realization) and the ones suggested by W3C recommendations (consistency, entailment, and conjunctive query answering) to introduce various nonstandard services, which we believe are almost indispensable

for practical use, with the obvious example being the various services for explaining and debugging ontologies (See section 5 for more details).

An orthogonal dimension to these services is the language they are implemented for. Pellet covers all of OWL-DL including inverse and transitive properties, cardinality restrictions, datatype reasoning for an extensive set of built-ins as well as user defined simple XML schema datatypes, enumerated classes (a.k.a, nominals) and instance assertions. The latter two are particularly key as a lot of the information published on the Semantic Web is instance heavy, contrary to traditional practice in the Description Logic community.

Just as Pellet does far more than is strictly required of a complete OWL consistency checker, it also does more than what is required of an OWL syntax checker. Pellet will automatically apply heuristics to an OWL-Full document to see if it can be coerced into an OWL-DL document and then processed in the normal way.

Finally, merely having a comprehensive range of services, without clear and easy ways to access them, is pointless. From the start, we have tried to make Pellet's services easily available to all sorts of users. We have a HTML form based service published on our website that casual dabblers can use to test their ontologies or queries. That service replicates the functionality available from Pellet's command line interface. We bundle Pellet inside our OWL ontology editor, Swoop [10], which, itself, can be run from a Web browser via Java WebStart. We also, as mentioned above, support the DIG interface and a panoply of Java based APIs for accessing Pellet's functionality.

To be a practical OWL-DL reasoner, one must balance functionality and accessibility. Pellet provides both.

3 Pellet Architecture and Design

Pellet, in its core, is a Description Logic reasoner. However, unlike other DL reasoners, it has been designed to work with OWL right from the beginning. This design choice had huge influence on the overall architecture. It affected how the tableaux reasoner was implemented, e.g. with the ability to reason with instance data (ABox reasoning) without making the Unique Name Assumption (UNA), and what kind of supporting modules to have, e.g. having an XML Schema datatype reasoner and a query engine.

The main design goal of Pellet was to have a small core reasoning engine that is suitable for extensions. Having a small core engine enabled us to develop interfaces for different RDF/OWL toolkits, such as Jena and the WonderWeb OWL API or to support applications that communicate through DIG interface. Even the core

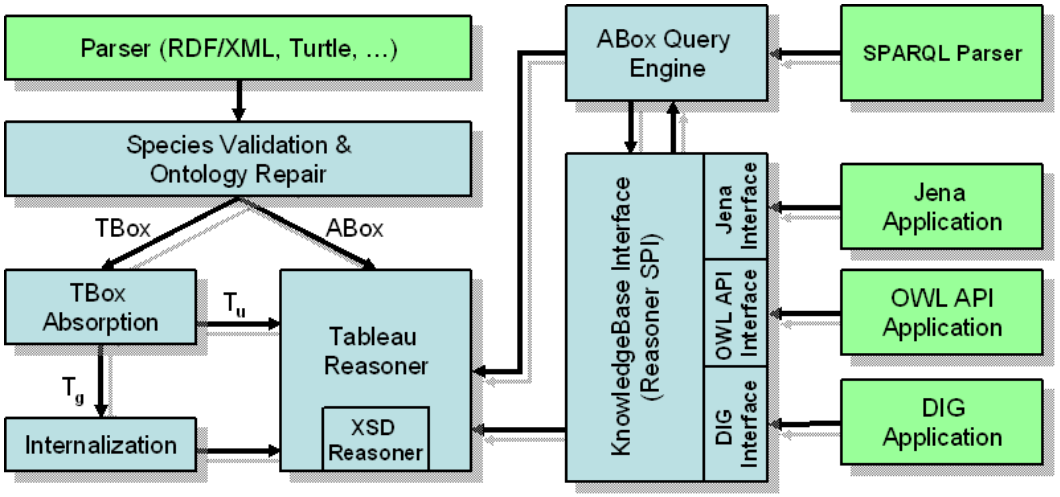

Fig. 2. Main components of the Pellet reasoner

engine itself is designed with extensibility in mind allowing us to implement extensions such as explanation generation for ontology debugging (see Section 4.2), multi-ontology reasoning using $\mathcal{E}$-Connections (see Section 5.1), non-monotonic reasoning with the epistemic operator (see Section 5.3) and integration with rule formalisms (see Section 5.2).

Figure 2 shows the main components of Pellet. The core of the system is the tableaux reasoner that checks the consistency of a knowledge base. The reasoner is coupled with a datatype oracle that can check the consistency of conjunctions of (built-in or derived) XML Schema simple datatypes. The OWL ontologies are loaded into the reasoner after species validation and ontology repair. This step ensures that all the resources have an appropriate type triple (a requirement for OWL-DL but not OWL-Full) and missing type declarations are added according to some heuristics (see Section 4.1 for details). During the loading phase, axioms about classes are put into the TBox component and assertions about individuals are stored in the ABox component. TBox axioms go through the standard preprocessing of DL reasoners, e.g. normalization, absorption and internalization, before they are fed to the tableaux reasoner. The system provides a thin layer for programmatic access through the Service Programming Interface (SPI) that provides convenience functions to access the reasoning services provided.

In what follows, we describe the main modules of the system in more detail.

3.1 Parsing and Loading

Pellet provides various different interfaces for loading ontologies. Pellet itself does not have an RDF/OWL parser but is integrated to different RDF/OWL toolkits that provide a parser. Ontologies represented in the data structures of such toolkits can be directly loaded to Pellet. Pellet also implements the *reasoner interfaces* defined in those toolkits to answer queries.

Each toolkit has quite different structures for representing OWL ontologies, e.g. Jena has a triple-oriented view where WonderWeb OWL API uses a view more akin to the OWL abstract syntax tree [11]. Therefore, Pellet includes different submodules that can load ontologies from these different representations. Loading ontologies from the Jena toolkit is done by examining the triples in the RDF graph and transforming them into OWL facts and axioms. This process also involves species validation and repair. The WonderWeb parser, on the other hand, already generates such structures in its own representation. Thus, loading an ontology simply involves traversing those structures with an appropriate visitor pattern.

In addition to the RDF/OWL support, Pellet also supports the standards developed for DL systems. The DIG (DL Implementors Group) interface [4] defines an HTTP-based *Tell/Ask* mechanism (with an XML syntax) for interacting with DL reasoners. Pellet supports the DIG interface and hence ontologies can also be loaded by communicating through an HTTP connection. Finally, Pellet has a parser to read the files written in KRSS format [12], a Lisp-like syntax traditionally used by DL reasoners. Using this parser, ontologies described in KRSS format can be directly loaded.

3.2 Tableaux Reasoner

The tableaux reasoner has only one functionality: checking the consistency of an ontology. According to the OWL model-theoretic semantics [2], an ontology is consistent if there is an interpretation that satisfies all the facts and axioms in the ontology. Such an interpretation is called a *model* of the ontology. The tableaux reasoner searches for such a model through a process of *completion*. The tableaux completion starts by constructing an initial completion graph from the ABox. The nodes in the completion graph intuitively stand for individuals and literals. Each node is associated with its corresponding types. Property-value assertions are represented as directed edges between nodes. The reasoner repeatedly applies the tableaux expansion rules until a clash (i.e. a contradiction) is detected in the label of a node, or until a clash-free graph is found to which no more rules are applicable.

All other reasoning tasks can be defined in terms of consistency checking. For example, checking whether an individual is an instance of a concept or not can be

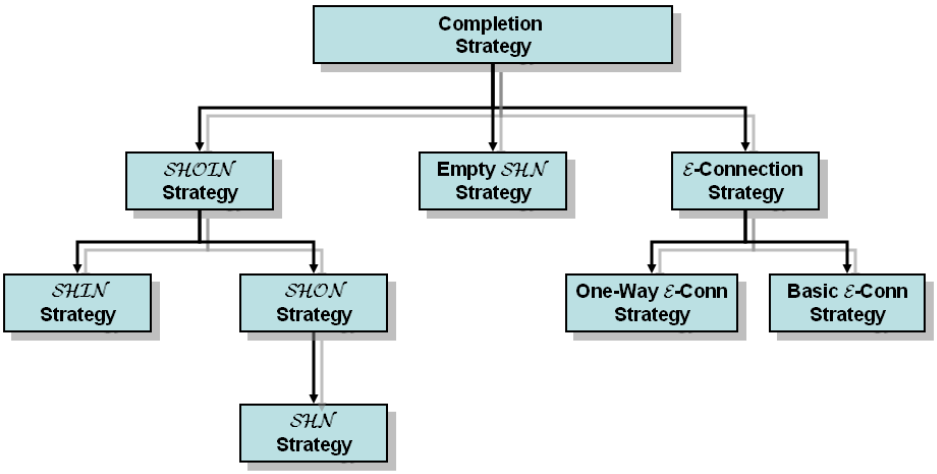

Fig. 3. Different completion strategies implemented in Pellet

tested by asserting that the individual is an instance of the complement of that class and then checking for (in)consistency.

In order to support future extensions, the internals of the tableaux reasoner are built on an extensible architecture. The completion algorithm inside the tableaux reasoner is designed so that different completion strategies can be plugged in. This approach has two major advantages: Different completion strategies with different heuristics can be used based on the characteristics of the given KB, e.g. the expressivity; also extensions such as $\mathcal{E}$-Connection support can be implemented without changing the rest of the system.

Figure 3 shows the different completion strategies currently implemented in Pellet. *$\mathcal{SHOIN}$Strategy* is the default completion strategy that supports the full expressivity of OWL-DL. This strategy is based on the recently developed decision procedure for $\mathcal{SHOIQ}$[3] [13].

The *$\mathcal{SHOIN}$Strategy* covers the full expressivity of OWL-DL and exhibits a good "pay as you go" behavior, e.g. the tableaux rule for nominals is never applied if there are no nominals in the KB. However, the blocking strategy[4] required for $\mathcal{SHOIN}$ (dynamic double blocking strategy) is quite complex and may not prevent the completion graph from getting very large. If it is known that there are no nominals in the KB then an optimized version of double blocking [16] can be used. Also, in this case, we do not even need to check if nominal rules is applicable (since it will never be) and save some more time. The *$\mathcal{SHIN}$Strategy* does exactly this, and hence, whenever the expressivity of the KB is detected to fall into

[3] $\mathcal{SHOIQ}$ is equivalent to OWL-DL extended with qualified cardinality restrictions which were present in the DAML+OIL language but omitted in OWL

[4] Blocking ensures the termination of tableaux algorithm by halting the completion process when a "cycle" that can cause infinite expansion is detected [14]. Several blocking strategies have been developed for different expressivities (see, for example, [15]).

this category, this strategy will be selected over the default $\mathcal{SHOIN}$*Strategy*. Similarly, the $\mathcal{SHON}$*Strategy* employs an even more efficient blocking strategy (subset blocking) and is selected whenever appropriate.

Some completion strategies behave quite different than others. For example, if there are no instances in the KB (just class and property descriptions) then it is known that every concept satisfiability check will start with a completion graph that has just one node. In such cases, more efficient completion strategies, e.g. the trace method, can be used. Moreover, when there are no inverse properties, we can use additional optimizations such as caching the satisfiability status of internal nodes. The *Empty*$\mathcal{SHN}$*Strategy* uses this approach and manages to handle very large KB's such as the famous Galen medical ontology.

The dynamic completion strategy selection ensures the soundness and completeness of the reasoner while exploiting the most efficient algorithm for the given KB.

3.3 Datatype Reasoner

The datatype reasoner is responsible for checking if the intersection of (possibly negated) datatypes is consistent or not. Datatypes in OWL are described using XML Schema which provides a rich set of simple datatypes including various numeric types (integers and floats), strings, and date/time types. In addition to this, XML Schema also provides several mechanisms for creating new types out of the base types, e.g. it is possible to define a type that consists of integer values less than or equal to 10 and integer values greater than 20. An intersection of datatypes is inconsistent when they have no data value in common, e.g. the intersection of `xsd:positiveInteger` and `xsd:negativeInteger` is empty.

Datatype reasoning in Pellet is based on the framework presented in [17]. This approach allows combining expressive DLs with an arbitrary type system. In this approach, the datatype reasoner is used as an oracle by the tableaux reasoner. For each literal node in the completion graph, the tableaux reasoner uses the datatype reasoner to determine if the intersection of all the datatypes associated with that node is satisfiable or not. Pellet's datatype reasoner supports all the built-in XSD types along with any type derived from numeric or date types.

3.4 Knowledge Base Interface

All the reasoning tasks can be reduced to a KB consistency test with an appropriate transformation. However, such transformations are not always trivial and doing a consistency check for every arbitrary query is very expensive. The System Programming Interface (SPI) of Pellet provides generic functions to manage

such transformations and hide the details from users. This *KnowledgeBase* interface makes decisions as to when to check the consistency of the ABox (if any changes have been made after the last time), when to classify all the concepts or when to realize all the individuals.

The *KnowledgeBase* interface provides functionality to answer arbitrary atomic queries. These queries can be related to classes (e.g. `getSubClasses`, `getDisjoints`, etc.), to properties (`getSubProperties`, `isFunctional`, etc.), or to individuals (e.g. `getTypes`, `getPropertyValues`, etc.). For boolean queries, the query is transformed to an unsatisfiability problem. For queries where a set of answers need to be returned, multiple consistency checks are required in theory but some optimizations are possible. For example, if we want to find all the instances of a concept, *KnowledgeBase* first computes all the obvious instances (e.g. all individuals explicitly asserted to be instances) and then uses sophisticated methods (see Section 6 for details) to find non-instances. For the remaining individuals, methods like binary instance retrieval [18] is used to test multiple individual with one consistency test. This general strategy is used for all the other queries, e.g. finding all the property values for a specific individual, because in the end all the other tests are reduced to similar unsatisfiability problems.

The *Knowledge Base interface*, as the rest of internal components, is built on the ATerm library [19]. ATerm (short for Annotated Term) is an abstract data type designed for the exchange of tree-like data structures between distributed applications. The ATerm library provides maximal subterm sharing and automatic garbage collection making it very suitable for representing complex OWL class expressions. The term sharing feature reduces the overall memory consumption spent for storing concept expressions and makes it is easy to transform the data from Pellet SPI to external APIs.

3.5 ABox Query Engine

The KnowledgeBase interface is coupled with an ABox Query Engine that answers conjunctive queries. This module supports queries written in the SPARQL [9] language as well as in the RDQL [8] language. More specifically, any ABox query expressed in these languages is parsed into an internal query format using the parser provided by HP Lab's ARQ module in the Jena toolkit. A query written in one of these RDF query languages is an ABox query if it satisfies the following conditions:

- No variable is used in the predicate position.
- Each property used in the predicate position is either a property (object or datatype) defined in the ontology or one of the following built-in properties: `rdf:type`, `owl:sameIndividualAs`, `owl:differentFrom`.

- If `rdf:type` is used in the predicate position, a constant URI is used in the object position.

Queries written in a more logic-oriented language such as KIF naturally fall into this category as the syntax does not allow violating these conditions.

The query answering for expressive DLs, such as OWL-DL, has some interesting implications. For example, a query that has *undistinguished variables*, i.e. variables that appear in the query body but not in the *SELECT* clause, need to be answered in a different way [20]. This is due to the fact that constructs such as `owl:someValuesFrom` cause new individuals to be created that are generally not to be included in the query result.

The Pellet query engine uses the "rolling-up" technique [20] to answer queries with undistinguished variables. This technique creates one concept expression from the query expression and reduces the problem to retrieving the instances of that concept. This step needs to be repeated for each distinguished variable. We have developed several optimizations to reduce the number of instance retrieval operations (See Section 6 for more details).

Figure 4 shows the general design of the query engine. There are multiple query engines that actually answer queries and one main query engine that preprocesses the query and selects the appropriate query engine. The first step of the main engine is to analyze the query and determine if it consists of independent sub-queries. If this is the case, the query is split into multiple queries which are answered separately. The results are combined at the end on a tuple by tuple basis. The next step is to examine the structure of the query and sort the patterns or variables to improve efficiency. The heuristic used here is to first bind variables with a smaller number of likely candidates, e.g. variables used with classes that have fewer instances or vari-

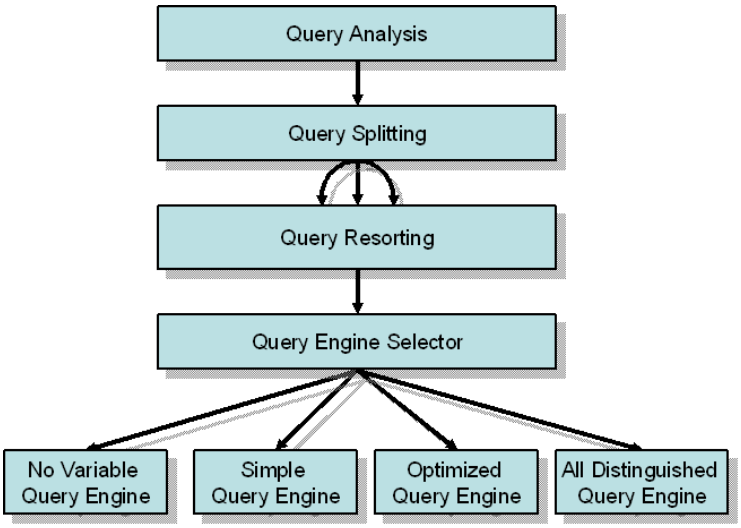

Fig. 4. Components of the query engine

ables used in conjunction with functional properties. This step is generally mixed with the last step, where one of the actual query engines generates the answer. For example, if there is no selection statement in the query, i.e. it is a boolean query, then no reordering is required, since the query can be answered in one rolling-up step.

4 Ontology Analysis and Repair

In this section we briefly describe the capabilities of Pellet for detecting *syntactic* and *semantic* defects in ontologies.

4.1 Species coercion

OWL has two major dialects, OWL-DL and OWL-Full, with OWL-DL being a subset of OWL-Full. All OWL knowledge bases are encoded as RDF/XML graphs. OWL-DL imposes a number of restrictions on RDF graphs, some of which are substantial (e.g., that the set of class names and individual names be disjoint) and some less so (that every item have an `rdf:type` triple). Ensuring that an RDF/XML document meets all the restrictions is a relatively difficult task for authors, and many existing OWL documents are nominally OWL-Full, even though their authors intend for them to be OWL-DL. Pellet incorporates a number of heuristics to detect "DLizable" OWL-Full documents in order to "repair" them.

The heuristics implemented in Pellet attempt to guess the correct type for an untyped resource. These are mainly standard operations, e.g. a resource used in the predicate position is inferred to be a property. Some situations have more than one solution, e.g. an untyped resource used only in one cardinality restriction can be any of object or a data property. In these cases, Pellet heuristics choose object properties and classes over data properties and datatypes by default, but this behavior can be configured.

Ensuring the vocabulary separation, e.g. disjointness of classes, properties and individuals, is another hard problem especially in the distributed Web environment where people might be required to import an OWL-Full ontology that they might have no control over. In such a case, it is not acceptable for a reasoner to reject processing the ontology altogether. For this reason, Pellet provides several options to the users where vocabulary separation is not respected:

- Ignore the statements that cause the problem. If a URI is used both as a class and as a property, one of these definitions will be ignored and the accepted definition depends on the order the statements are processed (this order is generally non-

deterministic and based on which underlying parser is used).

- Accept all the definitions for the URI but treat them differently for query answering. For example, if the same URI is defined both as a class and as a property, Pellet will create both a class and a property and associate the axioms with the corresponding definition. Depending on the queries, asking subclasses vs. asking sub properties, the appropriate definition will be used.
- Reject processing the ontology completely.

These options give the user more control about how to deal with the different cases and provide a plausible solution for a certain set of OWL-Full ontologies. On the other hand, some features of OWL-Full ontologies are completely out of scope for Pellet. For example, defining cardinality restrictions on transitive properties causes undecidability. Extending built-in vocabulary, e.g. creating a subproperty of `rdf:type`, requires a completely different reasoning procedure. Therefore, for such OWL-Full features only options provided are *Ignore* or *Fail*.

4.2 Debugging Support

As descriptions in OWL ontologies become more complicated, finding the cause of semantic errors, i.e., contradictions in ontological definitions, becomes an extremely hard task even for experts. Typically, reasoners only *detect* unsatisfiable concepts (or inconsistent ontologies); however, the diagnosis and resolution of the bug is not supported at all. To overcome this problem, Pellet contains two debug-

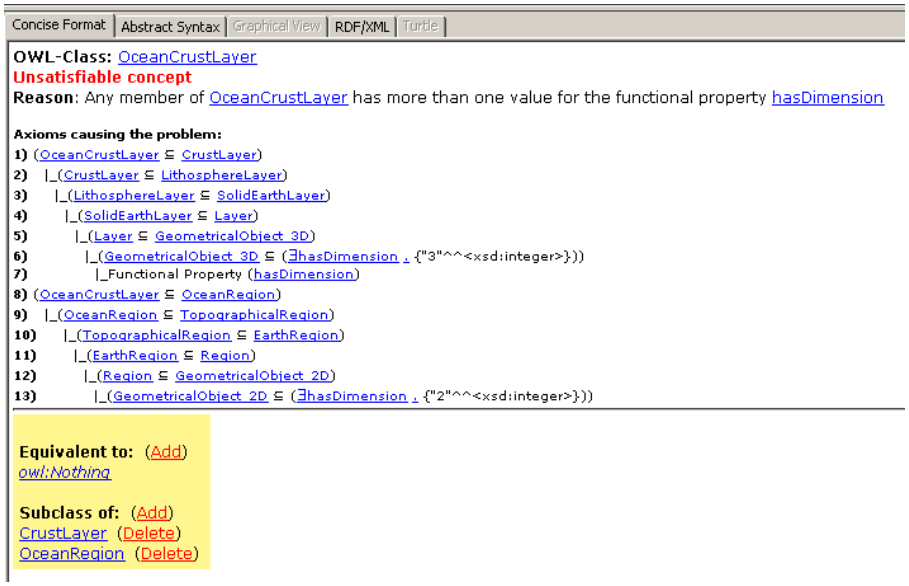

Fig. 5. The explanation of unsatisfiability for class `OceanCrustLayer` includes a description of the clash created in the tableaux reasoner. Also, the set of axioms responsible for this clash is extracted by the Pellet axiom tracing service. (Screenshot taken from the Swoop Ontology editor running Pellet)

ging services that help explain *why* the inconsistency occurs: the first service – *clash detection* is used to pinpoint the root contradiction or clash in the completion graph; and the second – *axiom tracing* is used to extract the relevant source axioms from the ontology responsible for the clash (see Figure 5 for an example). These services are used in the OWL Ontology Editor, Swoop [10], as a debugging aid for ontology users and modelers. [21]

5 Beyond OWL-DL

5.1 Multi-Ontology Reasoning using $\mathcal{E}$-Connections

$\mathcal{E}$-Connections [22] are a framework for combining several families of decidable logics, such as Description Logics, Modal Logics, as well as some logics of time and space. In an $\mathcal{E}$-Connection, the coupling between the combined logics is loose enough for obtaining general results about transfer of decidability: if reasoning is decidable in each of the logics in the combination, then it is decidable in the combined formalism as well. Thus, $\mathcal{E}$-Connections are computationally more robust than other combination methods, such as *Fusions* [23] or *Multidimensional Modal Logics*, [24], in which the interaction between the combined formalisms is closer and general transfer of decidability results cannot be expected.

A knowledge base in the combined language is composed of a set of knowledge bases, expressed in any of the component logics. The component KBs are interpreted over *disjoint* logical domains and connected by means of *link relations*. The new operators provided by the $\mathcal{E}$-Connection language are associated to the link relations and hence are used to describe the relationships between the connected KBs.

In [25] and [26] we have proposed tableau algorithms for different $\mathcal{E}$-Connection languages involving Description Logics. The basic strategy to extend a DL tableau algorithm with $\mathcal{E}$-Connections support is based on "coloring" the completion graph. Nodes of different "colors", or sorts, correspond to different domains (ontologies). The application of the expansion rules, blocking conditions and clash triggers depend on both the "color" of the node under consideration and the expressivity allowed on the link relations. When implementing tableau algorithms for $\mathcal{E}$-Connections as an extension of an OWL reasoner, all these issues must be thoroughly considered. For a detailed discussion on combined tableau algorithms for $\mathcal{E}$-Connections we refer the reader to [25] and [26].

Pellet has been extended with tableau-based decision procedures for several $\mathcal{E}$-Connection languages. The implementation is still in "beta" stage, but our initial experimental results show that the performance for the $\mathcal{E}$-Connected KBs is very

similar to their OWL counterparts. $\mathcal{E}$-Connections do not seem to affect existing optimizations or degrade the performance of the reasoner, even in our "naive" implementation. Currently, we are tuning the reasoner and performing a more extensive evaluation.

Finally, it is worth emphasizing here that, although Pellet currently can only handle $\mathcal{E}$-Connections of OWL-DL ontologies, it can be easily extended to other interesting $\mathcal{E}$-Connection languages, such as $\mathcal{E}$-Connection languages including the qualitative spatial logic $\mathcal{RCC}$-8 as a component logic, or Distributed Description Logics [27], which can be seen as sub-formalisms of basic $\mathcal{E}$-Connections [22].

5.2 Integration with Rules formalisms

The Semantic Web Rules Language (SWRL) [28] has recently been proposed as the basic rules language for the Semantic Web.

SWRL is based on a simple idea, namely, that the coupling between a DL Knowledge Base $\mathcal{K}$ and a Datalog program $\mathcal{P}$ is achieved by allowing the use of classes, object and datatype properties defined and used in $\mathcal{K}$ (called DL-atoms) in the Datalog rules in $\mathcal{P}$.

Unfortunately, although SWRL provides useful new expressive power, it is known to be undecidable.

In the mid and late 90s, many proposals for combining DLs and Datalog were presented under the name of "Hybrid Systems". The most prominent ones are $\mathcal{AL}$-Log [29] and CARIN [30]. Both can be seen as *decidable* subsets of SWRL [5] . In particular, in $\mathcal{AL}$-Log the only DL atoms allowed in the Datalog rules are classes and these can only be included in the body of the rule.

We have implemented a prototype of $\mathcal{AL}$-Log using Pellet. The implementation slightly generalizes the original $\mathcal{AL}$-Log in two ways: first, we use $\mathcal{SHOIN}(\mathbf{D})$ instead of $\mathcal{ALC}$, which was the language originally used in the DL component, and secondly we allow the use of OWL datatypes, and SWRL built-ins in the antecedent of Datalog rules.

The reasoner computes answers to queries based on the specification of both components and is based on the notion of *constrained SLD-derivation* and *constrained SLD-refutation*, as presented in [29]. The system has been implemented in Prolog, coupled to Pellet. Potential applications for $\mathcal{AL}$-Log include Web Policies and Web Services.

[5] If only unary and binary predicates are allowed

5.3 Non-monotonic Reasoning with the Epistemic Operator

Non-monotonic logics have been generally successful in capturing several forms of common sense and database reasoning. A prominent family of non-monotonic formalisms are rooted in various forms of the *closed world* assumption. Sometimes, it is reasonable to assume that the information at hand is *complete*. Under these circumstances, if a formula cannot be proved true or false in a KB, it is considered to be false. This constitutes the *closed-world assumption* (CWA). Under the *open world assumption* (OWA), on the other hand, it is accepted that the knowledge in our KB is perhaps incomplete and hence, if a formula cannot be proved true or false, we do not draw any conclusion.

It would be desirable to be able to "turn on" the closed-world assumption when needed in OWL in order the reap the benefits of nonmonotonicity, but without giving up OWL's open-world semantics in general. The logic $\mathcal{ALCK}$ allows for this interaction. Many useful nonmonotonic features such as integrity constraints and procedural rules (among others– see [31]) are formalizable in this logic.

In [31], the epistemic operator **K** is added to the Description Logic $\mathcal{ALC}$. The **K** operator allows queries that assume the CWA, making $\mathcal{ALCK}$ a nonmonotonic formalism. The **K** operator (which is a kind of necessity operator) can be applied to a concept or role.

In its simplest form, $\mathcal{ALCK}$ allows the use of **K** operator *only* in queries and assume that queries that are posed to an $\mathcal{ALC}$ knowledge base Σ. Intuitively, the query $\langle a : \mathbf{K}C \rangle$ (resp. $\langle a\mathbf{K}Rb \rangle$ for a role) is read as "Is the individual a *known to be* C?"

Pellet includes an implementation of the $\mathcal{ALCK}$ language. In addition to the **K** queries outlined above (accessible as an extension to the SPARQL query language), we also admit a restricted use of **K** in the terminology, in the form of an *epistemic rule*. An epistemic rule is of the form of $\mathbf{K}C \sqsubseteq D$ where C and D are $\mathcal{ALC}$ concepts. In order to allow for **K** and epistemic rules we have extended the KRSS format (as it is not clear how to integrate such rules with OWL's RDF/XML syntax).

6 Implementation and Optimizations

Expressive Description Logics, such as $\mathcal{SHOIN}(\mathcal{D})$, are known to have very high worst-case complexity. As a consequence, there exists a significant gap between the design of a decision procedure and the achievement of a practical implementation. Naive implementations are doomed to failure.

In order to achieve acceptable performance, modern DL reasoners implement a suite of optimization techniques. These optimizations lead to a significant improvement in the performance of the reasoner and have proved effective in wide variety of realistic applications.

Pellet implements most of the state of the art optimization techniques provided in the DL literature and have been implemented in other systems such as FaCT++ and RACER (see [32] for a complete description) including:

- *Normalization and Simplification* Normalization is the process of transforming all the concepts in a form where contradictions involving complex concepts are detected early during tableaux expansion. Simplification detects obvious clashes during the normalization process and also gets rid of redundant elements of a concept expression.
- *TBox Absorption* is a technique that tries to eliminate GCIs as possible from a KB by replacing them with primitive definitions. Absorption can be improved if general axioms are absorbed into domain or range axioms whenever possible [33].
- *Semantic Branching* When a disjunction is being expanded, a single disjunct D is chosen and two possible search trees are obtained by adding either D or $\neg D$. Because the two search trees are strictly disjoint, there is no possibility of wasted search.
- *Dependency-directed Backjumping* Dependency-directed backjumping eliminates unproductive backtracking search by finding which branching points are responsible for a clash and jumping back over the intervening branching points without exploring any alternatives.
- *Oldest-first Disjunction Selection* When an individual contains many disjunctions in its label, the order in which these disjuncts are expanded has an importance. To improve the effectiveness of dependency-directed backjumping we first expand the disjunct that depends on the minimum branch number.
- *Caching Intermediate Satisfiability Status* During a consistency check there may be many fresh nodes created in the completion graph. Some of these nodes can be very similar and caching the satisfiability status of these nodes can save significant computation time especially if there are no inverse properties or nominals used in the ontology.
- *Optimized Blocking* Performance can be improved drastically by optimizing the double blocking strategy so that cycles are detected earlier and completion graphs do not become very deep. Pellet incorporates this blocking strategy for the KBs with suitable expressivity, e.g. $\mathcal{SHI}$.
- *Top-Bottom Search for Classification* Classification performance is highly improved when an algorithm based on the traversal of concept hierarchy is used instead of checking the subclass relation between each named class. Using the asserted subclass relations and exploiting the transitivity of the subclass relation during traversal proves to be very useful.
- *Model Merging* The obvious non-subsumption relation between two concepts

can be detected by inspecting the cached models for the concepts. If merging the model for C and $\neg D$ does not have any possible clash then we can infer that C is not a subclass of D without doing the expensive consistency test.

The optimizations above have been implemented and proved useful for the Description Logic $\mathcal{SHIN}(\mathcal{D})$ (a.k.a OWL-Lite). However, from an implementation point of view, the recent achievement of a decision procedure for $\mathcal{SHOIN}(\mathcal{D})$ poses new challenges:

- While many optimization techniques are completely independent of the DL supported by the reasoner, others are valid for certain logics only. In particular, some optimizations for reasoning with ABoxes, e.g. *chain contraction* [34], are not applicable in the presence of nominals. Moreover, in the presence of nominals, ABox assertions can affect concept satisfiability and TBox classification. In other words, nominals break the traditional "separation" between TBox and ABox in Description Logics. As a consequence, ontologies with nominals in the TBox and large number of instances in the ABox are likely to compromise the performance of DL reasoners.
- Nominals are not supported by state of the art DL reasoners, except for Pellet. Thus, there is very little experience in developing techniques for dealing with nominals efficiently in practice. In particular, to the best of our knowledge, no optimizations specific for nominals, other than ours, have been explored.

We have developed a suite of new optimizations for facing these challenges in the presence of individuals. We provide a brief description of these optimizations (details will be provided in our upcoming technical report):

- *Nominal Absorption* Pellet uses an improved absorption technique where axioms involving enumerations (tt oneOf) are absorbed into ABox assertions.
- *Partial Backjumping* During backjumping it is highly likely that some useful information generated a the intervening branching points is being thrown away. Partial backjumping inspects the dependency set information to keep this information and avoids repeated application of same tableaux rules.
- *Learning-based Disjunct Selection* When there are large number of individuals in the KB with similar characteristics, it is highly likely that selecting the same disjunct from a disjunction will work on all the individuals. Learning-based disjunct selection keeps track of the successful disjunct selections and when a disjunction is being expanded it always selects the disjunct that caused less clashes in previous applications.
- *Nominal-based Model Merging* Nominals always have a fixed interpretation in the domain. Pellet exploits this property to improve the model merging algorithm for detecting obvious non-subsumptions.

In addition, Pellet incorporates several optimizations for ABox query answering. Different techniques are employed depending on the structure of the query. The

rolling-up technique [20] provides an easy reduction of query answering to KB consistency, but it is important to use pseudo models as much as possible to avoid expensive consistency tests. Examining the query atoms without rolling up helps to achieve this since the pseudo models for named concepts are generally available (or generating them is profitable considering they are used very frequently). It is possible to minimize the number of candidates for each variable and reduce the required consistency tests. With this approach, the queries with only distinguished variables can be answered without any consistency tests most of the time. Similar to dynamic completion strategy selection, Pellet uses a combination of different techniques to ensure completeness and best efficiency. The different query strategies are preferred in the following order:

- *No distinguished variables*: First check if all ground atoms are trivially entailed (without performing a consistency test). Then pick one constant from the query, roll everything else into one concept and return true as the answer if the selected individual is an instance of the rolled up concept.
- *Only distinguished variables*: Pick a variable with minimum number of possible bindings (or start with a constant if there is one), verify that all the types given in the query actually hold, get the values for the related variables and continue with that variable.
- *One distinguished variable*: Using pseudo model checking compute possible bindings for the variable, roll the query into one concept and apply binary instance retrieval using the candidates.
- *More than one distinguished variable*: Similar as above to compute candidates; variables are then sorted based on the number of candidates. Different permutation of bindings are applied to the query and rolling up is used to verify if the query is entailed. Bindings for variables are added one by one allowing to prune a large number of permutations when a certain binding fails.

In practice, it turns out that most queries fall into the second category ("`SELECT *`" queries). Moreover, generally, users desire this kind of query evaluation in order to query based on something akin to the Closed World Assumption. For this reason, Pellet provides an option to turn off special treatment of undistinguished variables.

7 System Evaluation

In this section, we evaluate the performance of the reasoner for the tasks of consistency checking, classification, realization, and conjunctive query answering. All the experiments have been performed on a Pentium Centrino 1.6GHz computer with 1.5GB memory. The maximum memory amount allowed to Java was set to 256MB for each experiment. All the timings presented in this section are computed as the average of 10 independent runs.

Name	Expressivity	Triples	Cla. / Prop. / Ind.	Load	Cons.	Clas.	Real.	Total
AKT	$\mathcal{ALCHOIF}(\mathcal{D})$	1771	169 / 137 / 75	0.21	0.02	0.73	0.11	1.08
Tambis	$\mathcal{SHIN}(\mathcal{D})$	4200	392 / 100 / 0	0.12	0.00	1.10	0.01	1.23
SUMO	$\mathcal{ALH}(\mathcal{D})$	3688	630 / 238 / 435	0.25	0.00	1.11	0.11	1.48
Food	$\mathcal{ALCOF}$	869	64 / 8 / 45	0.27	0.05	0.66	0.58	1.57
OWL-S	$\mathcal{SHOIF}(\mathcal{D})$	2463	121 / 187 / 283	0.27	0.02	1.19	0.62	2.09
Financial	$\mathcal{ALCIF}$	65723	59 / 16 / 17941	2.38	2.19	0.03	2.73	7.34
SWEET	$\mathcal{SHOIF}(\mathcal{D})$	5894	1400 / 137 / 110	0.21	0.01	7.95	0.39	8.56
Wine	$\mathcal{SHOIF}(\mathcal{D})$	2708	137 / 17 / 206	0.11	1.30	10.21	1.10	12.71
Galen	$\mathcal{SHF}$	30854	2749 / 413 / 0	1.30	0.00	47.16	0.00	48.46

Fig. 6. Pellet performance on commonly used ontologies of varying complexity and expressivity. All times are shown in seconds. Links to these ontologies can be found at the web page http://www.mindswap.org/2003/pellet/performance.

Figure 6 lists the loading, consistency checking, classification and realization timings for some well-known OWL ontologies: the Food/Wine example from the OWL guide [35], the portal ontology from the AKT project, the Congo example from the OWL-S coalition, the SWEET set of ontologies from NASA, the SUMO upper ontology and, finally, the medical ontologies Tambis and Galen.

These results show that Pellet has a reasonable performance for these ontologies of varying complexity and expressivity. It takes substantial amount of time to classify the Galen medical ontology which is expected for an ontology of this size. It is interesting to note that processing the small Wine ontology takes longer than other larger ontologies. The reason is that the Wine ontology was created to showcase all the features in OWL with a heavy emphasis on nominals. This caused the axioms in the ontology to be very complex and the ontology is a challenging case even for incomplete reasoners. Sound and complete reasoning Pellet provides for this ontology is still faster compared to the incomplete reasoners.

Figure 7 shows the classification times of different DL reasoners for some ontologies included in the standard DL benchmark test suite [36]. The systems we have compared with Pellet (version 1.3) are Racer Pro 1.8.2 [6] , the commercialized version of the Racer system, and FaCT++ 0.99.5, a C++ reimplementation of the FaCT reasoner. The selected ontologies do not contain ABox assertions, nominals or qualified number restrictions [7] The results are sorted based on the size of the ontologies.

[6] This version was not released as of the writing but the Racer developers provided a preview version for our experiments since there were significant improvements, especially for query answering, compared to the latest release

[7] Qualified cardinality restrictions are not present in OWL-DL and not supported by Pel-

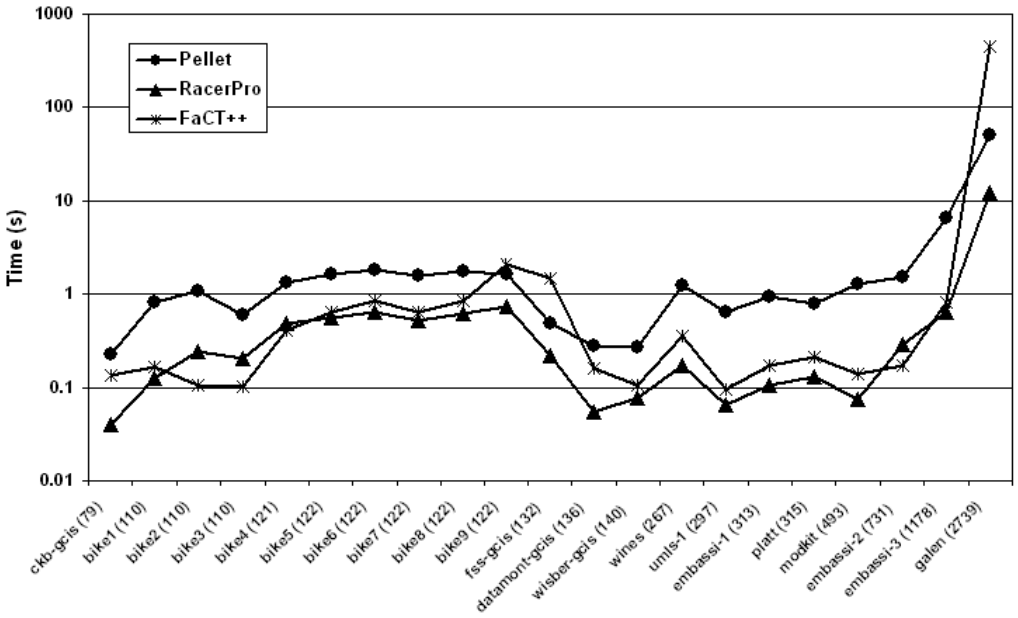

Fig. 7. Results for the test cases in DL benchmark suite

The scale is logarithmic to improve the presentation.

The results show that Pellet is not as efficient as RacerPro or FaCT++ for classification but still has acceptable performance. The practical difference in performance seen in these cases is typically not very significant for applications. It is also interesting to note that, even though FaCT++ usually significantly performs better than Pellet, for the complex Galen medical ontology Pellet is 9 times faster.

Finally, we have evaluated the performance of conjunctive ABox query answering. Since FaCT++ does not provide reasoning support for instances, we have only compared Pellet with RacerPro. For this experiment, we have used the data and queries included in the Lehigh University Benchmark (LUBM) [37]. The data generator in the benchmark creates information about universities, departments, professors, students and courses. In order to provide a finer grained comparison, we used three different data sets which contain 1, 3 and 5 universities, respectively. The number of instances in these data sets are 17174, 55664, and 102368.

In our experiments, we have evaluated three different features: the time it takes the reasoner to check the consistency of the data, the amount of total preparation time spent before queries can be answered (excluding parsing and loading time), and finally the time it takes to answer each query. Figure 8 summarizes the results we obtained on three data sets. The results show that Pellet significantly outperforms RacerPro for consistency checking (also note that RacerPro was unable to perform

let. ABox reasoning is not supported by FaCT++ and nominals are not available either in FaCT++ or RacerPro.

Consistency Checking Time (s)			
System	LUBM(1,0)	LUBM(3,0)	LUBM(5,0)
Pellet	2.26	11.82	22.69
RacerPro 1.8.2	13.02	125.72	Out of Mem.

Preparation Time (s)			
System	LUBM(1,0)	LUBM(3,0)	LUBM(5,0)
Pellet	2.35	11.92	22.85
RacerPro 1.8.2	4.05	25.57	56.63

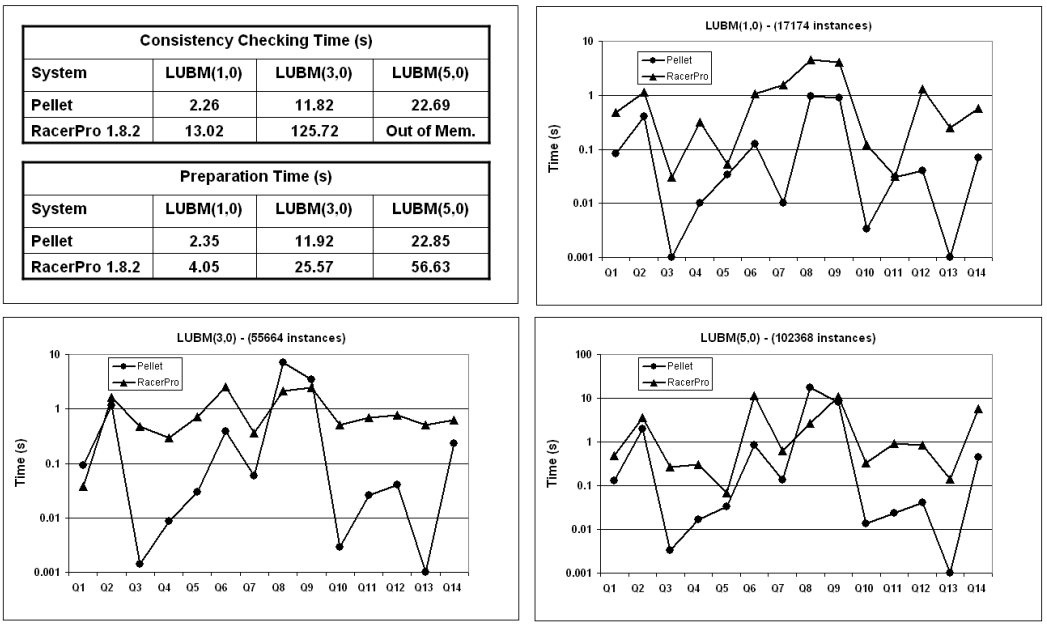

Fig. 8. Results for conjunctive query answering in LUBM

the consistency test for 15 universities where Pellet spent only 22 seconds). For this reason, we had to disable consistency checking for RacerPro for query answering. Even without consistency checking RacerPro has a much higher initial preparation time compared to Pellet (RacerPro builds index structures in order to perform efficient query answering). Note that preparation time for Pellet is very close to consistency checking time because there is no other significant operation that Pellet performs for initialization.

The query answering times for three data sets show that Pellet generally performs better compared to RacerPro. Query answering times do not change for most of the queries as the data size increase, with the exception of query 8 and 9. It is also important to note that RacerPro has two different query answering modes where the faster mode, the mode we used in these experiments, is not complete with respect to more expressive ontologies. On the other hand, the techniques Pellet uses for query answering is based on consistency checking and logically sound and complete for more expressive ontologies. In the future, we are planning to do a more through analysis with ontologies of varying expressivity.

These experiments show that Pellet is not as efficient as FaCT++ or RacerPro in TBox reasoning tasks. However, its performance is still competitive for real-world applications. We also see that Pellet performs well when reasoning with large number of instances.

8 Conclusion and Future Directions

In this paper, we have presented Pellet, an open source OWL-DL reasoner with a number of unique features. Pellet exhibits a competitive performance and has already been used in both industry and academia.

In the near future, we are planning to extend the reasoner in six main directions.

(1) *More Expressive DLs*: Many applications demand a Description Logic beyond OWL-DL. In particular, qualified number restrictions and extended datatype representation and reasoning, such as multi-arity datatype predicates [38] and inverse-functional datatype properties, are especially relevant.
(2) *New optimizations*, including secondary-storage support for reasoning with large number of individuals, query answering for ontology management, and novel optimization techniques based on *partitioning* of OWL ontologies [39] and axiom tracing.
(3) *Combination with other logical formalisms*. Many Semantic Web applications, such as multi-media systems, require the ability to reason with space, time and motion. We are currently working on extending Pellet with various spatio-temporal representation and reasoning functionalities.
(4) *Incremental reasoning*: Many applications, such as Ontology Management, OWL-based multimedia systems or task computing, involve repeated changes in OWL KBs in a relatively short period of time. For these applications, it is critical for the reasoner to recompute as little as possible after each update.
(5) *Rules*: We will continue to evolve our support of rules toward full SWRL support (focusing on interesting decidable subsets such as DL Safe rules along the way) and investigate other ways of combining rules with OWL. We intend this work to follow and support the forthcoming W3C working group on rules.
(6) *Non-monotonicity*: We plan to extend our support for the **K** (and related **A**) operator as well as investigating such extensions as defaults, integrity constraints, and various forms of closed world reasoning.

Our initial goal with Pellet was modest and pragmatic: We wanted OWL to become a W3C Recommendation and we wanted a "real" reasoner we could both use in our applications and extend at will. OWL is now a Recommendation and Pellet is a mature, practical, *accessible* tool. We have found that having (intelligible) source code goes a long way to demystifying description logic theorem proving. While there is plenty of room for improving Pellet's performance both by straightforward engineering and with new optimizations, it has reached a level that is acceptable for most use. Indeed, given the utility of Pellet's unique functionality, it is either the only choice (e.g., if one uses nominals), or the only choice for everything except final deployment in a production environment, and even there, it is a good choice.

9 Acknowledgments

The authors would like to especially thank Edna Ruckhaus who implemented the $\mathcal{AL}$-Log coupling as well as Ron Alford and Michael Grove for their contributions to the code. We are also grateful to Fujitsu Labs of America and especially to Ryusuke Masuoka for providing real-world problems that required new levels of functionality and performance. Finally, we would like to thank Ian Horrocks, Peter Patel-Schneider and all the people who have contributed to the Pellet mailing list for helpful discussions and suggestions.

References

[1] J. J. Carroll, J. D. Roo, OWL Web Ontology Language Test Cases, W3C Recommendation http://www.w3.org/TR/owl-test/ (2004).

[2] P. Patel-Schneider, P. Hayes, I.Horrocks, OWL web ontology language Abstract Syntax and Semantics, W3C Recommendation http://www.w3.org/TR/owl-semantics/ (2004).

[3] F. Baader, W. Nutt, Basic description logics, in: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003, pp. 43–95.

[4] S. Bechhofer, R. Möller, P. Crowther, The DIG description logic interface, in: Proc. of the Int. Description Logics Workshop (DL 2003), 2003.

[5] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: Implementing the semantic web recommendations, in: Proc. of the 13th Int. World Wide Web Conference (WWW 2004), 2004.

[6] S. Bechhofer, P. Lord, R. Volz, Cooking the semantic web with the OWL API, in: Proc. of the 2nd Int. Semantic Web Conf. (ISWC 2003), Sanibel Island, Florida, 2003.

[7] I. Horrocks, P. Patel-Schneider, Reducing OWL entailment to description logic satisfiability, J. of Web Semantics 1 (4) (2004) 345–357.

[8] A. Seaborne, RDQL - A query language for RDF, W3C Submission http://www.w3.org/Submission/RDQL/ (2004).

[9] E. Prud'hommeaux, A. S. (editors), Sparql query language for rdf, W3C Working Draft (21 July 2005) http://www.w3.org/TR/rdf-sparql-query/ (2005).

[10] A. Kalyanpur, B. Parsia, J. Hendler, A tool for working with web ontologies, International Journal on Semantic Web and Information Systems 1 (1).

[11] S. K. Bechhofer, J. J. Carroll, Parsing OWL DL: Trees or triples?, in: Proc. of the 13th Int. World Wide Web Conference (WWW 2004), 2004.

[12] P. F. Patel-Schneider, B. Swartout, Description logic specification from the KRSS, http://www.bell-labs.com/user/pfps/papers/krss-spec.ps (2003).

[13] I. Horrocks, U. Sattler, A tableaux decision procedure for SHOIQ, in: Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005), Morgan Kaufman, 2005.

[14] F. Baader, U. Sattler, An overview of tableau algorithms for description logics, Studia Logica 69 (2001) 5–40.

[15] I. Horrocks, U. Sattler, S. Tobies, Practical reasoning for very expressive description logics, Logic Journal of the IGPL 8 (3) (2000) 239–263.

[16] I. Horrocks, U. Sattler, Optimised reasoning for $\mathcal{SHIQ}$, in: Proc. of the 15th Eur. Conf. on Artificial Intelligence (ECAI 2002), 2002, pp. 277–281.

[17] I. Horrocks, U. Sattler, Ontology reasoning in the $\mathcal{SHOQ}(\mathcal{D})$ description logic, in: Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI 2001), 2001, pp. 199–204.

[18] V. Haarslev, R. Möller, Optimization techniques for retrieving resources described in OWL/RDF documents: First results, in: Proc. of the 9th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2004), 2004, pp. 163–173.

[19] M. van den Brand, H. de Jong, P. Klint, P. Olivier, Efficient annotated terms, Software, Practice and Experience 30 (3) (2000) 259–291.

[20] I. Horrocks, S. Tessaris, Querying the semantic web: a formal approach, in: Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002), 2002, pp. 177–191.

[21] A. Kalyanpur, B. Parsia, E.Sirin, J. Hendler, Debugging unsatisfiable concepts in OWL ontologies, Journal on Web Semantics, 2005, To Appear.

[22] O. Kutz, C. Lutz, F. Wolter, M. Zakharyaschev, $\mathcal{E}$-Connections of Abstract Description Systems, Artificial Intelligence 156(1):1-73.

[23] F. Baader, C. Lutz, H.Sturm, F.Wolter, Fusions of description logics and abstract description systems, Journal of Artificial Intelligence Research (JAIR), 16:1-58.

[24] D. Gabbay, A. Kurucz, F. Wolter, M. Zakharyaschev, Many-Dimensional Modal Logics: Theory and Applications, Vol. 148 of Studies in Logic, Elsevier, 2003.

[25] B. C. Grau, B. Parsia, E.Sirin, Combining OWL ontologies using $\mathcal{E}$-connections, Journal on Web Semantics, 2005, To Appear.

[26] B. Cuenca-Grau, Combination and integration of ontologies on the semantic web, Ph.D. thesis, Universidad de Valencia (2005).

[27] A. Borgida, L. Serafini, Distributed description logics: Assimilating information from peer sources, Journal of Data Semantics, 1:153-184.

[28] I. Horrocks, P. F. Patel-Schneider, A proposal for an OWL rules language, in: Proc. of the 13th Int. World Wide Web Conference (WWW 2004), 2004, pp. 723–731.

[29] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, $\mathcal{AL}$-Log: Integrating datalog and description logics, Journal of Intelligent Information Systems 10 (1998) 227–252.

[30] A. Levy, M.-C. Rousset, CARIN: A representation language combining horn rules and description logics, Artificial Intelligence 104 (1-2) (1998) 165–209.

[31] F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, W. Nutt, An epistemic operator for description logics, Artificial Intelligence 100 (1-2) (1998) 225–274.

[32] I. Horrocks, Implementation and optimisation techniques, in: F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, 2003, pp. 306–346.

[33] D. Tsarkov, I. Horrocks, Efficient reasoning with range and domain constraints, in: Proc. of the Int. Description Logic Workshop (DL 2004), 2004, pp. 41–50.

[34] V. Haarslev, R. Möller, An empirical evaluation of optimization strategies for abox reasoning in expressive description logics, in: Proc. of the Int. Description Logics Workshop (DL'99), 1999, pp. 115–119.

[35] M. Smith, C. Welty, D. McGuiness, OWL Web Ontology Language Guide, W3C Recommendation http://www.w3.org/TR/owl-guide/ (2004).

[36] I. Horrocks, P. F. Patel-Schneider, DL systems comparison, in: Proc. of the Int. Description Logics Workshop (DL'98), 1998, pp. 55–57.

[37] Z. P. Yuanbo Guo, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, Journal of Web Semantics 3 (2) (2005) 158–182.

[38] J. Z. Pan, I. Horrocks, Extending Datatype Support in Web Ontology Reasoning, in: Proc. of the 1st Int. Conf. on Ontologies, Databases and Applications of SEmantics (ODBASE 2002), 2002, pp. 1067–1081.

[39] B. Cuenca-Grau, B. Parsia, E. Sirin, A. Kalyanpur, Automatic partitioning of OWL ontologies using $\mathcal{E}$-connections, in: Proc. of the Int. Description Logics Workshop (DL 2005), 2005.