Konclude: System Description

Andreas Steigmiller[a,*], Thorsten Liebig[b], Birte Glimm[a]

[a] *Institute of Artificial Intelligence, University of Ulm, Ulm, Germany*
[b] *derivo GmbH, Ulm, Germany*

Abstract

This paper introduces Konclude, a high-performance reasoner for the Description Logic $\mathcal{SROIQV}$. The supported ontology language is a superset of the logic underlying OWL 2 extended by nominal schemas, which allows for expressing arbitrary DL-safe rules. Konclude's reasoning core is primarily based on the well-known tableau calculus for expressive Description Logics. In addition, Konclude also incorporates adaptations of more specialised procedures, such as consequence-based reasoning, in order to support the tableau algorithm. Konclude is designed for performance and uses well-known optimisations such as absorption or caching, but also implements several new optimisation techniques. The system can furthermore take advantage of multiple CPU's at several levels of its processing architecture. This paper describes Konclude's interface options, reasoner architecture, processing workflow, and key optimisations. Furthermore, we provide results of a comparison with other widely used OWL 2 reasoning systems, which show that Konclude performs eminently well on ontologies from any language fragment of OWL 2.

Keywords: Web Ontology Language, Description Logics, Tableau Theorem Proving

1. Introduction

The current version of the Web Ontology Language (OWL 2) [1] is based on the very expressive Description Logic (DL) $\mathcal{SROIQ}$ (see [2] for a DL introduction) and extends the first version of OWL with more expressive language features such as qualified cardinality restrictions and property chains.

Many existing reasoning systems have been adapted to OWL 2 and several new optimisations have been developed to deal with the latest language features or specific profiles. Despite all the progress, reasoning performance still shows up as a noticeable issue for users. Because of the N2EXPTIME-complete worst-case complexity for standard reasoning tasks in $\mathcal{SROIQ}$ [3], this is expected at least for some ontologies. However, there are several clues which indicate that further improvements are possible. For instance, an effective coupling of fully-fledged OWL 2 reasoning procedures with tractable procedures for OWL 2 profiles could improve the overall performance. Moreover, multi-core computers are ubiquitous now, but state-of-the-art reasoners for expressive DLs do not yet implement an effective parallelised processing architecture.

In this system description, we introduce the novel reasoning system Konclude,[1] which addresses both aforementioned issues. It incorporates different reasoning procedures and implements new as well as extensions of existing optimisations adapted to concurrent processing within a multi-core, shared memory architecture. This significantly improves the running time of reasoning tasks for many real-world ontologies.

As of now, Konclude handles the DL $\mathcal{SROIQ}$ and also supports nominal schemas [4] which generalise arbitrary DL-safe rules [5]. Konclude supports the most common reasoning services such as classification, realisation, queries for sub-classes, class instances or types of individuals and it can be used as server or via command line on various platforms.

The rest of this system description is organised as follows: we next introduce the system's architecture; in Section 3 we give an overview of the integrated optimisations; in Section 4 we present the result of a comprehensive evaluation and comparison to other state-of-the-art reasoners before we conclude in Section 5.

*Corresponding Author
Email addresses: andreas.steigmiller@uni-ulm.de (Andreas Steigmiller), liebig@derivo.de (Thorsten Liebig), birte.glimm@uni-ulm.de (Birte Glimm)

[1] Available at http://www.konclude.com/

2. System Architecture

Konclude is implemented in C++ and makes use of the cross-platform application framework Qt.[2] The reasoner runs on all Qt supported platforms including Windows, OS X, Linux, and Solaris.

Konclude offers two communication options: First, it is an OWLlink server that exposes ontology management and reasoner functionality to one or more clients (usually ontology-based applications) via the W3C OWLlink protocol [6]. As OWLlink server, Konclude supports OWL 2 XML as content format with HTTP as transport protocol as specified in the OWLlink HTTP/XML binding.[3] Since OWLlink is an implementation-neutral and extensible protocol, it is well-suited for complex and distributed semantic applications that aim for independent system components on dedicated machines in order to achieve overall stability as well as scalability. Ontology-aware applications that use the OWL API can easily link to Konclude via the OWLlink OWL API Adapter.[4] Depending on the communication volume, the latter may result in some performance drain caused by the additional serialisation and parsing overhead in comparison to native OWL API implementations. Second, users can interact with the reasoner via a command line interface, which can, for example, be used for testing or benchmarking. The latter is similar to those of other OWL reasoners and allows for loading an ontology (in OWL 2 XML syntax or as OWLlink request), executing a basic reasoning request with a particular system configuration, and optionally for saving any results or OWLlink response.

2.1. Reasoner Management

The main components of Konclude and their interaction is shown in Figure 1. The overall workflow for handling ontologies and reasoning requests can be divided into three main processing stages: parsing, loading, and reasoning. Some processing stages are divided into several steps (e.g., loading consists of building the internal representation of the ontology and preprocessing the axioms). Each processing step is usually controlled by a manager, e.g., the preprocessing manager.

Konclude can simultaneously handle several ontologies and all ontologies are processed lazily, i.e., only those steps are performed, which are necessary to answer a given request for a certain ontology. In that respect, parsing of ontology axioms does not immediately trigger the creation of an internal representation nor the preprocessing of these axioms. Instead, the parsed axioms (possibly from different clients) are first collected in containers in order to keep track of the different revisions of an ontology.

As depicted in Figure 1, the `Reasoning Manager` is a key component of Konclude for the handling of requests that require reasoning. Such requests are characterised by a list of conditions that have to be satisfied in order to generate an answer. The `Reasoning Manager` is then responsible for identifying and managing the processing workflow that is necessary to satisfy the conditions of these requests. For instance, if the user requests the class hierarchy of an ontology, then it is necessary to build the internal representation, to preprocess these data structures, to test the consistency of the ontology and, finally, to classify. Furthermore, reasoning tasks, such as classification, might trigger several other precomputing steps, such as the construction and population of internal data structures and caches, to make reasoning more efficient. Further delegation to sub-managers is stopped by the `Reasoning Manager` if required processing steps cannot be successfully completed. For instance, classification is initialised only if the ontology is first proved to be consistent. The workflow of a particular request depends on the characteristics of its respective ontology. For example, for a deterministic ontology the `Classification Manager` creates a `Deterministic Classifier`.

2.2. Reasoning Procedures

At its core, Konclude is based on the tableau calculus for $\mathcal{SROIQ}$ [7]. Tableau calculi are extensible, sound, and complete decision procedures for expressive DLs and are used in many well-known reasoning engines. More precisely, they are refutation-based calculi that transform reasoning tasks into one or more consistency problems. The objective of the tableau decision procedure is to check whether the consistency problems can be satisfied or not. For that purpose the calculus tries to systematically construct an abstract model of the satisfiability input – the so-called *completion graph*. Roughly speaking, a completion graph is a graph of nodes related by edges. Each node represents an individual in a model of the ontology and is labelled with a set of concepts (called classes/class expressions in OWL) that this individual has to satisfy. Edges relate nodes with successor nodes and are labelled with roles (called properties/property expressions in OWL). The tableau algorithm uses a set of expansion rules to syntactically decompose concepts in node labels, where each such rule application can add new concepts to node labels and/or

[2] `http://qt-project.org/`

[3] `http://w3.org/Submission/owllink-httpxml-binding`

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

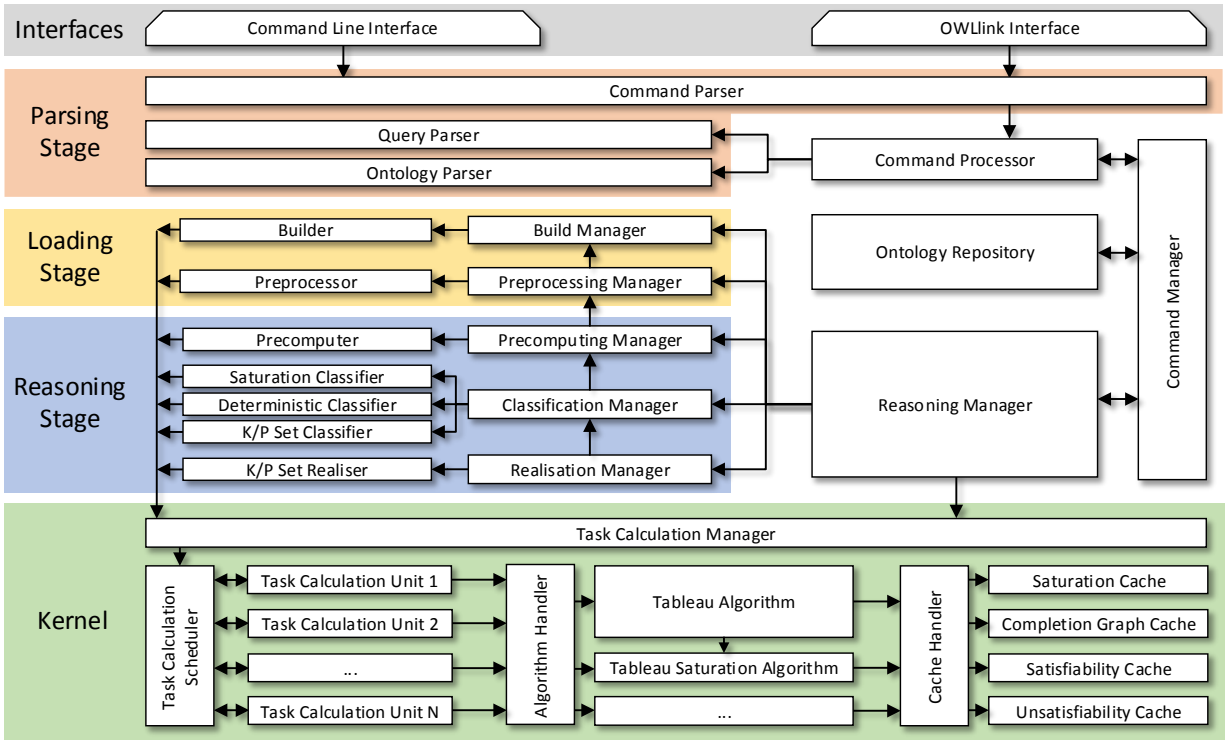

Figure 1: Architecture and important components of Konclude

new nodes and edges to the completion graph, thereby explicating the structure of a model. The rules are repeatedly applied until either the graph is fully expanded (no more rules are applicable) or a contradiction (called a clash) is discovered. If the rules (some of which are non-deterministic) can be applied such that they build a fully expanded, clash-free completion graph, then the checked consistency problem can be satisfied. Termination of the algorithm is ensured by a technique called *blocking* which prevents the generation of infinite completion graphs for cyclic concepts, i.e., for concepts that are defined with axioms such that they directly or indirectly refer to themselves. For example, if an axiom states that each *Person* has a father that is also a *Person*, then the class *Person* is cyclic.

In addition to the tableau calculus, Konclude incorporates more specialised reasoning procedures. Most notably, it utilises a variant of completion-/consequence-based saturation procedure [8, 9] that is adapted to the data structures of the tableau algorithm.

2.3. Parallel Processing Levels

Konclude allows for parallelisation at three different levels of its processing architecture:

Parallel Ontology Processing Konclude is able to process several ontologies concurrently. For instance, it can classify several ontologies at the same time since the classifiers and many other important system components are instantiated in separate, independent threads. This is especially useful if Konclude is run as an OWLlink server to serve multiple clients that operate on different ontologies. In order to avoid inter-process bottlenecks, communication among main components is based on peer-to-peer messages.

Parallel Query/Sub-Query Answering Konclude is able to answer several queries for one particular ontology at a time. In case these queries depend on each other (such as realisation or classification) they will also benefit from already finished processing steps or cached (intermediate) results from previous requests. Furthermore, one particular query (such as satisfiability of all classes, classification, realisation) can be naturally split up into sub-queries (e.g. concept subsumption or satisfiability). These sub-queries are scheduled for concurrent processing and will incrementally contribute to the globally available result caches.

Parallel Completion Graph Processing The most sophisticated parallelisation mechanism in Konclude is on

the completion graph level. Konclude allows for concurrent processing of alternative branches caused by non-deterministic tableau rules, i.e., non-deterministic alternatives such as disjuncts of a disjunction or merges caused by an at-most cardinality restriction. However, special mechanisms are required in order to control this kind of concurrency. As an example, in case of a successful expansion of one alternative its sibling branches (and sub-branches thereof) need to be discarded.

The number of worker threads used by Konclude is adjustable by the user on start up. This is not only useful for fair benchmarking with single-threaded systems, as done in our cross evaluation, but also has an effect on memory consumption since each worker has to set up and maintain some exclusive working memory. Consequently, reasoning with more worker threads usually increases the memory consumption.

As of now, parallelisation in Konclude is applied to the tableau procedure as well as on the overall system level. Other processing steps such as preprocessing, precomputation, and saturation are currently not carried out in parallel.

2.4. Task Scheduling

As mentioned in the previous section, Konclude offers parallelisation at different processing levels and at various levels of user control. Whereas the potential increase of concurrency at ontology level (cf. 2.3) is just a factor of the number of distinguished ontologies, the increase at query or completion graph level easily becomes exponential. Therefore, the latter ones are managed by a dedicated component within the `Kernel` of Konclude, namely the `Task Calculation Manager`. Instead of creating a new thread for every task, the `Kernel` has access to a configurable number of threads, namely its `Task Calculation Units`. The number of `Task Calculation Units` (or workers) does not necessarily have to match the number of CPU cores, but the latter typically is a practical upper bound for scalability from parallelisation.

The `Task Calculation Scheduler` is a vital and highly optimised `Kernel` component that distributes tasks to the different `Task Calculation Units`. In case the number of tasks exceeds the number of available `Task Calculation Units` – either initially or due to sub-tasks generated in consequence of non-determinism – the remaining tasks are buffered temporarily for later allocation by the `Task Calculation Scheduler`. Furthermore, the `Task Calculation Scheduler` also has to take care of certain task interdependencies. For example, the classifiers in Konclude typically create several basically independent subsumption tests. Some of the latter may require to branch non-deterministically which in turn schedules a multitude of new sub-tasks. The `Task Calculation Scheduler` ensures a balanced assignment of tasks to `Task Calculation Units`, where more high-level and independent tasks are delegated with higher priority to different `Task Calculation Units`. This helps with keeping communication between threads small.

The `Task Calculation Units` can exchange information over caches, whereby also intermediate results can be shared. Likewise to other key infrastructure, Konclude relies on its own, highly-optimised cache implementation that ensures that any cache is accessible without significant blocking, which is of fundamental importance for parallel read operations.

Moreover, the `Kernel` uses a specialised memory management, which allocates memory in larger blocks and releases these blocks if the corresponding tasks are not required anymore. This is especially useful in a parallelised environment, where memory is often allocated and released by different threads, in order to avoid blocking and defragmentation of memory.

The `Task Calculation Units` are typically used to construct completion graphs with the tableau algorithm. However, the `Task Calculation Units` can also be used to process certain tasks with other algorithms (e.g., saturation of concepts) and, as a result, all the CPU-intensive work from the processing steps can be delegated to `Task Calculation Units`. Thus, the main work can be done within the `Kernel`, which makes the majority of the parallelisation controllable.

3. Optimisations

A naive tableau algorithm is not suitable for handling typical real-world ontologies since completion graphs can easily become very large and may contain many sources of non-determinism. To tackle these challenges, Konclude applies a significant range of state-of-the-art and new optimisations. The main conceptual and technical challenge of the system was to extend important optimisations to work with more expressive languages as well as to effectively implement them within an inherently parallel system architecture.

Optimisations within DL reasoning systems are typically associated with one of the three main phases of processing and are, therefore, divided into optimisations for preprocessing, consistency checking, and higher reasoning tasks. In the following, we provide a short overview about Konclude's most important optimisations for each of these categories.

Preprocessing Optimisations A well-known and very important optimisation of this kind is *absorption*, which aims at rewriting general concept inclusion (GCI) axioms to avoid non-determinism in the tableau algorithm. The partial absorption used in Konclude (see [10] for details) is an optimised version of the standard absorption, which further reduces non-determinism by also absorbing parts of concepts. This also allows for rewriting axioms that use more expressive language features such as at-most cardinality restrictions. The technique generalises other known absorption techniques such as *binary absorption* [11], *role absorption* [12] and *nominal absorption* [13]. Moreover, we use the absorption in Konclude to achieve a very efficient handling of nominal schemas [10]. Of course, we integrated also other standard preprocessing optimisations such as *lexical normalisation* and *simplification* to simplify the detection of clashes [2].

Consistency Checking Optimisations Checking the consistency of a problem is the core decision procedure of a tableau-based reasoner. Since these checks typically occur very often, we have optimised this task in particular. First, Konclude integrates many standard optimisations including *lazy unfolding*, *semantic branching*, *boolean constant propagation*, *anywhere blocking*, *dependency directed backtracking*, and *caching of satisfiability status* (see [2] for a more detailed descriptions of these optimisations). Many of these optimisations have been adapted to a parallelised environment, e.g., dependency directed backtracking requires that (sub-)tasks of irrelevant non-deterministic branches are cancelled even when these tasks are already distributed to other `Task Calculation Units`. Second, some optimisations have been extended significantly to achieve better performance and to work well with more expressive languages. For instance, caching has been extended to work with ontologies that use nominals and inverses. In contrast to traditional caching, Konclude also distinguishes between unsatisfiability and satisfiability caching, which allows for optimised retrieval from these caches. Moreover, Konclude uses a sophisticated dependency tracking mechanism that keeps track of how consequences have been derived. The tracked dependencies are then used to realise a *precise unsatisfiability caching* [14] and to cache required expansions for nodes in the completion graph, which allows for blocking nodes earlier in Konclude and works particularly well for mostly deterministic ontologies. Third, we have devised a range of new optimisations. Konclude supports, for example, the *reuse and caching of entire completion graphs*, which significantly reduces the reasoning time for ontologies with nominals after the first consistency check. Although also other reasoners reuse deterministic parts of the completion graph from the initial consistency check [13], the approach realised in Konclude is an extension in several ways. For instance, Konclude caches also those parts of completion graphs where non-deterministic choices have been made and employs a sophisticated mechanism to find cache matches. Once a completion graph is cached (e.g, after the initial consistency check), Konclude only has to process the nodes for those individuals for which the expansion to the cached versions of these individuals cannot be trivially ensured. Hence, also for non-deterministic ontologies with nominals, Konclude can significantly reduce the number of nodes that have to be processed in subsequent consistency checks. Furthermore, we have combined the tableau algorithm with a saturation-based reasoning technique [15], which is an adaptation of the completion-/consequence-based reasoning procedures [8, 9] as used for the OWL 2 EL profile. Since our saturation procedure is adapted to the data structures of the tableau algorithm, we achieve a very tight integration of the two reasoning techniques, i.e., even for non-EL ontologies Konclude can quickly obtain many sound inferences by combining both reasoning methods, while EL reasoners cannot be used or become incomplete as soon as some non-EL features (e.g., some disjunction) are used. Moreover, the coupling with the saturation procedure is used to block earlier in tableau-based consistency checks and to extract interesting information such as obvious subsumptions and non-subsumptions. Finally, the developed *pool-based merging* [14] reduces the non-determinism when handling cardinality restrictions. Together with many low level optimisations such as the *incremental completion graph building* of (non-deterministic) branches and *sophisticated processing queues*, consistency checking in Konclude is usually unproblematic in practice even for ontologies with high expressivity (e.g., inverse roles, nominals, and cardinality restrictions).

Higher Level Optimisations Higher level reasoning tasks (such as classification or realisation) are usually reduced to a multitude of consistency checks such that they benefit from the optimisations of the latter kind as much as possible. In addition, dedicated higher level optimisations aim at reducing the number of consistency checks. For classification and realisation this is achieved by a *known/possible set classification and realisation* approach [16], which is additionally supported by different *model merging* techniques [17]. Furthermore, the combination with the saturation-based reason-

Table 1: Statistics of ontology metrics for the evaluated ontology repositories (Ø stands for average and M for median)

Repository	# Ontologies	Axioms		Classes		Properties		Individuals		%	%	%	%
		Ø	M	Ø	M	Ø	M	Ø	M	DL	EL	QL	RL
Gardiner	276	6,143	95	1,892	16	36	7	90	3	59.4	15.9	17.8	23.9
NCBO BioPortal	403	25,561	1,068	7,617	339	47	13	1,782	0	61.5	33.7	28.0	18.9
NCIt	185	178,818	167,667	69,720	68,862	116	123	0	0	71.9	4.3	4.3	1.6
OBO Foundry	422	44,424	1,990	8,033	839	28	6	24,868	66	58.7	18.7	11.6	25.1
Oxford	383	74,248	4,249	8,789	544	52	13	18,798	12	85.9	20.6	13.8	22.7
TONES	200	7,697	337	2,907	100	28	5	66	0	87.0	37.5	26.5	21.0
Google Crawl	413	6,282	194	1,122	38	69	15	830	1	68.5	8.5	10.2	7.5
OntoCrawler	544	1,876	119	125	18	56	12	637	0	70.5	8.3	9.9	7.4
OntoJCrawl	1,680	5,847	218	1641	43	29	8	810	0	55.2	13.2	9.7	7.2
Swoogle Crawl	1,335	2,529	109	420	21	26	8	888	0	67.8	9.7	10.6	12.4
ALL	6,141	18,583	252	4,635	50	39	9	3,674	0	65.1	14.4	12.6	12.3

ing technique often allows a one-pass handling of those parts of ontologies that are in the OWL EL profile. In particular, this enables the very fast one-pass classification of EL ontologies (and similar less expressive ontologies).

4. Evaluation

In this section we provide an evaluation that compares the current version (0.5.0) of Konclude and the state-of-the-art reasoners FaCT++ 1.6.2 [18], HermiT 1.3.8 [19], Pellet 2.3.1 [20], and ELK 0.4.1 [21] (for EL ontologies). The evaluation uses a large test corpus of ontologies that have been obtained by collecting all downloadable and parseable ontologies from the Gardiner ontology suite [22], the NCBO BioPortal,[5] the National Cancer Institute thesaurus (NCIt) archive[6], the Open Biological Ontologies (OBO) Foundry [23], the Oxford ontology library,[7] the TONES repository,[8] and those subsets of the OWLCorpus [24] that were gathered by the crawlers Google, OntoCrawler, OntoJCrawl, and Swoogle.[9] We used the OWL API 3.4.5 for parsing and we converted all ontologies to self-contained OWL/XML files. For each of the 1,380 ontologies with imports, we also created a version with resolved imports and a version, where the import directives are simply removed (which allows for testing the reasoning performance on the main ontology content without imports, which are frequently shared by many ontologies). Since Konclude does not yet support datatypes, we removed all data properties and we replaced all data property restrictions with *owl*:*Thing* in all ontologies. Note that all reasoners were evaluated on the same modified ontologies. Table 1 shows an overview of our obtained test corpus with overall 6,141 ontologies including statistics of ontology metrics for the source repositories. Please note that there are many ontologies that are not in the OWL 2 DL profile, which is almost always due to undeclared entities. Parsers can usually fix such issues and there is no effect on the reasoning.

The evaluation was carried out on a Dell PowerEdge R420 server running with two Intel Xeon E5-2440 hexa core processors at 2.4 GHz with Hyper-Threading and 48 GB RAM under a 64bit Ubuntu 12.04.2 LTS. Each test was executed with a time limit of 5 minutes, but without any limitation of memory allocation. All reasoners were queried via the OWLlink protocol, which is natively supported by Konclude and we used the OWLlink OWL API Adapter for all other reasoners. In order to facilitate a comparison between the reasoners that is independent of the number of CPU cores, we first configured the parallelised reasoners ELK and Konclude to use only one worker thread (indicated with the suffix x1, e.g., Konclude 0.5.0 x1) and then we separately evaluated the effect of parallelisation.

The shown times do not include loading times. This is a disadvantage for Konclude since most reasoners do some preprocessing while loading, whereas Konclude only performs preprocessing on demand. This seems to be confirmed by the accumulated loading time, which is 1,400 s for Konclude, 4,130 s for FaCT++, 6,452 s for Pellet and 7,869 s for HermiT. However, even just parsing with the OWL API often requires more time in comparison to the built-in parser of Konclude.

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

[6] `http://ncit.nci.nih.gov/`

[7] `http://www.cs.ox.ac.uk/isg/ontologies/`; We ignored repositories that are redundantly contained in the Oxford ontology library (e.g., the Gardiner ontology suite).

[8] `http://owl.cs.manchester.ac.uk/repository/`

[9] In order to avoid too many redundant ontologies, we only used those subsets of the OWLCorpus which were gathered with the crawlers OntoCrawler, OntoJCrawl, Swoogle, and Google.

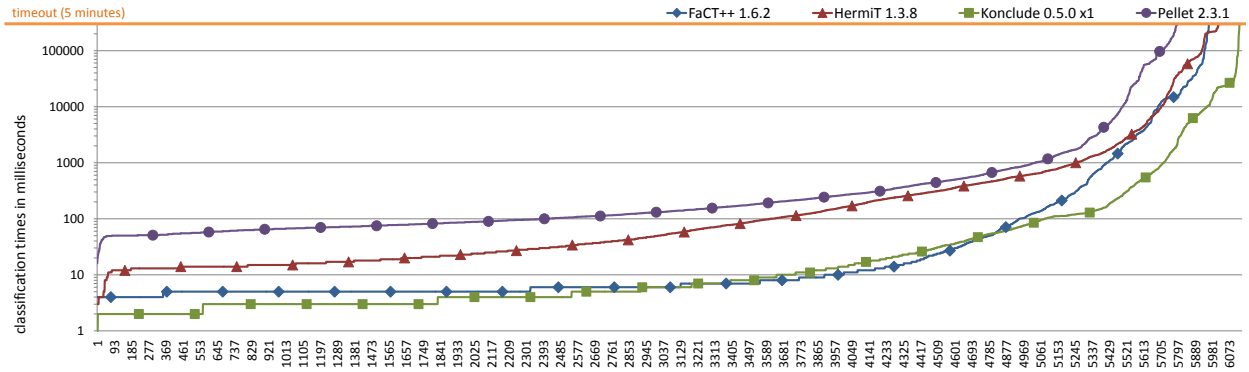

Figure 2: Comparison of separate sorted classification times for all ontologies in milliseconds

We further ignored all errors of other reasoners, e.g., if a reasoner failed to load an ontology or some of its axioms (e.g., due to unsupported axioms or irregular role inclusions) or produced fatal errors (e.g., program crashes, unhandled Java exceptions), we tried to continue the test. This usually means that the reasoner processed an ontology with less or even no axioms. Therefore, the runtime of a reasoner that produced many errors may seem to be better than they actually are.

4.1. Classification

Figure 2 shows an overview of the classification times for FaCT++, HermiT, Konclude (with one worker thread), and Pellet over all ontologies in our corpus. The classification times (shown at a logarithmic scale in milliseconds) are sorted in ascending order for all reasoners separately, i.e., the "easiest" ontologies for each reasoner are shown on the left side, whereas the "hardest" ontologies are shown on the right side. Not included are the times for loading and for outputting the class hierarchy. Due to Konclude's lazy processing approach, processing steps after parsing, such as the creation of internal data structures and preprocessing, are triggered with the classification request and, therefore, the times required for these steps are also included for Konclude.

Konclude processed all ontologies, but only approximated consequences of role inclusions if they were not regular as specified for OWL 2 DL. HermiT reported errors for 260, FaCT++ for 280, and Pellet for 318 ontologies. As indicated by the errors, these ontologies were usually not completely processed by Pellet, FaCT++, and HermiT. An often occurring error consisted of axioms that are used to declare individuals as different, but where only one individual was specified. In contrast, Konclude simply ignores such axioms and prints out a warning.

In summary, the overall classification performance of Konclude is better than that of FaCT++, HermiT, and Pellet. Although Konclude does not outperform the other reasoners on each ontology, it mostly dominates in the number of ontologies that could be classified in a given time limit. A particular strength of Konclude is the handling of very difficult ontologies, where the sophisticated optimisation techniques pay off. For example, the Brain Architecture Management System (BAMS) ontology from the Oxford ontology library can be classified by Konclude in 3.4 s, whereas all other reasoners timed out. Especially the comprehensive absorption technique in Konclude dramatically improves the performance for such ontologies. For other very complex ontologies also the detailed caching technique of Konclude is very important, which results, especially in combination with absorption, in better classification times. To give a few examples, the x-anatomy (Gardiner), the x-metazoan-anatomy (Gardiner), the composite-metazoan (Oxford), and the Ontology of Adverse Events (OAE) (OBO Foundry) can only be classified by Konclude within the given time limit. Moreover, due to the caching and reusing of entire completion graphs, Konclude shows significant improvements for ontologies that are intensively using nominals. For example, the well-known wine ontology can be classified by Konclude in 0.1 s, whereas the other reasoners require 2.5 – 6.3 s. It is also worth pointing out that a large amount of ontologies contain only a few axioms which do not fall into the OWL EL profile, e.g., by using language features such as inverse or functional object properties. For example, the Biomodels ontology, the Cell Cycle Ontology (CCO), and the Regulation of Gene Expression Ontology (ReXO) from the NCBO BioPortal are very big (500,000 – 850,000 axioms), but almost completely in the EL profile. For

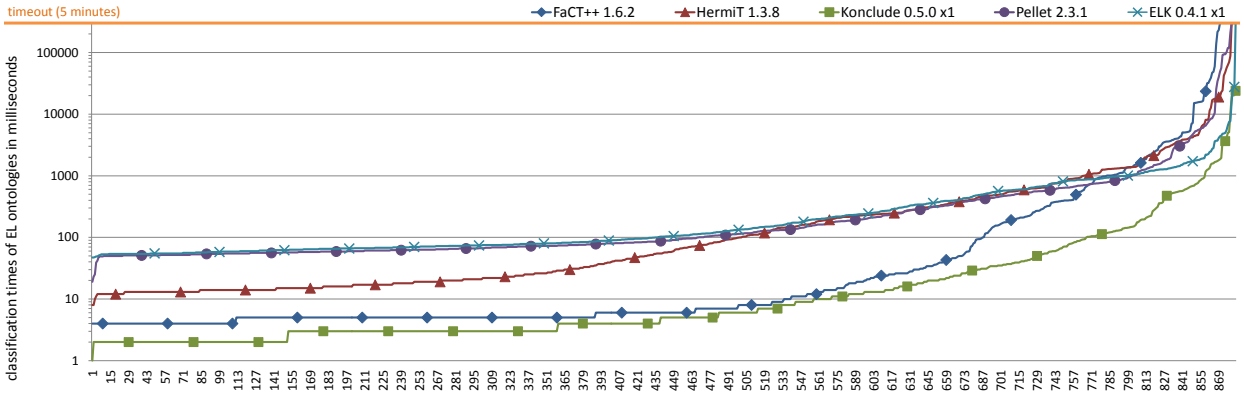

Figure 3: Comparison of separate sorted classification times for EL ontologies in milliseconds

such ontologies the tight coupling of the tableau- and the saturation-based methods in Konclude results in a very good pay-as-you-go behaviour and the ontologies can be classified in 33.5 s, 46.8 s, and 55.1 s, respectively. All other tableau-based reasoners failed to classify these ontologies within the time limit and OWL EL reasoners are incomplete. Furthermore, complex roles (e.g., due to property chains) are usually not causing significant problems for Konclude. In particular, the Data Mining OPtimization (DMOP) ontology from the OntoJCrawl of the OWLCorpus defines several symmetric and transitive roles in combination with role compositions (property chains in OWL), but can easily be classified by Konclude in 0.8 s, whereas HermiT already requires 142.9 s and the other reasoners completely fail to classify the ontology within the time limit.

Overall, Konclude reached the time limit only for 15 ontologies due to the wide range of optimisations, whereas HermiT timed out for 129, FaCT++ for 183, and Pellet for 355 ontologies. Moreover, Konclude was the fastest reasoner for the classification of 4,652 of the 6,141 ontologies in our corpus. In comparison, FaCT++ was the fastest for 1,362, HermiT for 69, and Pellet for 58 ontologies. Of course, it has to be noticed that there is a natural advantage for the C++ reasoners (FaCT++ and Konclude) since they have less overhead than Java implementations and many ontologies are so simple that hardly any reasoning is required.

Konclude has, however, also downsides. In particular, the handling of (big) cardinality restrictions still leaves room for improvement. For example, Konclude requires for the classification of the atom-complex-proton-2.0 ontology from the TONES repository 35.2 s, whereas Pellet only requires 11.4 s (FaCT++ and HermiT timed out). Especially the integration of algebraic methods, where cardinality restrictions are handled as a system of linear (in)equations [25], could provide significant improvements. Furthermore, for a few ontologies the saturation can hardly gain any useful information and its overhead causes a performance decrease. This is, for example, the case for the classification of the teleost-taxonomy ontology from the OBO Foundry, where Konclude requires 5.5 s, whereas the other reasoners only require 1.2 – 2 s.

The classification results of Konclude are identical with those of HermiT for all ontologies where Konclude and HermiT processed the ontology without reporting an error. There are, however, a few differences to FaCT++ and Pellet (and there are also differences between these reasoners) mainly for ontologies with irregular role inclusion axioms (HermiT refuses the processing of such ontologies). We analysed the results for such ontologies manually and, to the best of our knowledge, Konclude derived the correct subsumptions.

Figure 3 shows an overview of the classification times for FaCT++, HermiT, Konclude, Pellet, and ELK over all EL ontologies in our corpus. Again, the classification times are shown at a logarithmic scale and they are separately sorted for each reasoner. Please also note that the classification times for Konclude and ELK were obtained by using only one worker thread. Unsurprisingly, ELK performs well for many larger EL ontologies, but the current version of ELK was not able to classify the hierarchy_closure ontology from the Oxford ontology library. Furthermore, ELK required more time for the classification of many very small ontologies than the majority of the other reasoners, which is possibly also a consequence of a delayed preprocessing or due to the creation of additional threads within the Java environment. Only Konclude successfully classified all 881 EL ontologies within the time limit and it was also the fastest reasoner for 784 ontologies. However, there

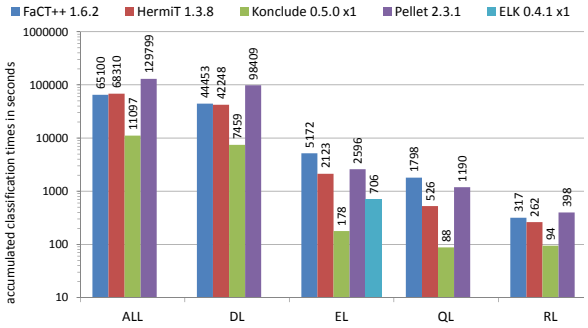

Figure 4: Comparison of the accumulated classification times in seconds grouped by the OWL 2 profiles

are some larger EL ontologies, where ELK is significantly faster than Konclude. This is possibly due to the fact that Konclude is designed as a $\mathcal{SROIQV}$ reasoner and has, therefore, a more comprehensive preprocessing mechanism (e.g., the axioms have to be absorbed and cannot be directly indexed), which possibly requires more time for some ontologies.

Figure 4 shows a comparison of the accumulated classification times for all ontologies and also for ontologies of the different OWL 2 profiles. Note that the accumulated classification times are again shown at a logarithmic scale, but now in seconds. The chart reveals that Konclude has the least accumulated classification times for all OWL 2 profiles with respect to the ontologies in our corpus. They are especially low for the EL, QL, and RL profile since Konclude did not reach the time limit for any ontology in these profiles.

4.2. Consistency

We also evaluated the reasoning task of consistency checking for all ontologies in our corpus with FaCT++, HermiT, Konclude, and Pellet. Again, we only measured the time for the consistency check and we used only one worker thread for Konclude. If we accumulate the times, we get 3,893 s for Konclude, 7,076 s for HermiT, 17,535 s for FaCT++, and 32,622 s for Pellet. Although Konclude has again the best overall performance and, in particular, the fewest number of timeouts, Konclude is also the slowest system for some ontologies. Again, this is often a consequence of the lazy processing paradigm in Konclude, which starts processing not before receiving the consistency checking request and, therefore, the reported times by Konclude also include the loading and preprocessing of ontologies.

4.3. Parallelisation

In addition, we evaluated the effects of the parallelisation. For that purpose, we run the classification reasoning tasks for all ontologies in our corpus also with two and four worker threads (i.e., `Task Calculation Units`) in Konclude. Two worker threads reduced the accumulated classification time from 11,097 s that were obtained without parallelisation to 9,980 s. Four worker threads further reduced the accumulated classification time to 9,249 s. Clearly, the gained speedup through parallelisation is far from linear, which is, on the one hand, a consequence of the hardware that disadvantages parallelisation by dynamically overclocking the processor if only few CPU cores are active[10] and, on the other hand, it has to be considered that the measured classification times also include unparallelised processing steps such as the building of the reasoner's internal ontology representation, preprocessing and several kinds of precomputing. Since many ontologies are relatively easy for Konclude due to the implemented optimisations, the time spent on the unparallelised processing steps often has a significant portion of the overall processing time. In particular, the saturation in Konclude is not yet parallelised and, therefore, increasing the number of worker threads does not significantly improve the overall performance for many EL-like ontologies. In contrast, there exists a range of ontologies where the parallelisation works very well. For example, the classification for the fmaOwlDLComponent_1_4_0 ontology from the TONES repository can be improved from 79.8 s with one worker thread to 45.5 s with two worker threads and to 27.1 s with four worker threads. Similar improvements are achieved for many expressive ontologies such as FMA, DOLCE, several variants of GALEN, and the majority of all NCI Thesaurus ontologies.

We also evaluated the effects of the parallelisation of ELK for the classification of all EL ontologies in our corpus. By increasing the worker threads to two, ELK was able to reduce the accumulated classification time from 706.2 to 637.1 s, and with four worker threads the accumulated classification time was further reduced to 591.2 s. Similarly to Konclude, the improvement factor from the parallelisation is not ideal, which probably relies on the same causes: unparallelised (pre-)processing steps and hardware influences.

5. Conclusions and Future Work

In this system description we have introduced Konclude, a new OWL 2 DL reasoner that supports the

[10] `http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`

DL $\mathcal{SROIQV}$. We have described Konclude's parallel processing architecture and the integrated key optimisation techniques that result in remarkable overall reasoning performance. In order to support the latter, we have presented and discussed results of a comprehensive comparison between Konclude and other state-of-the-art reasoners for different reasoning tasks and many ontologies. The comparison has revealed that Konclude performs eminently well for all evaluated reasoning tasks on all OWL 2 profiles. The bottom line is that Konclude significantly outperforms any other reasoner in terms of accumulated runtime over our test suite of over 6,000 ontologies for either classification or consistency checking. The good performance of Konclude has also been verified recently at the 2013 OWL Reasoner Competition (ORE 2013), where a previous development version of Konclude won 3 out of 9 offline benchmark categories [26].

Note that Konclude is still under development. Our plan is to work on the integration of new optimisation techniques and we will further extend parallelisation to more processing steps and more reasoning tasks. Moreover, we want to improve support for incremental reasoning and enhance the query options (e.g. conjunctive queries). The integration of OWL 2 datatypes, which will make Konclude a fully compliant OWL 2 DL reasoner, is already work in progress.

References

[1] W3C OWL Working Group, OWL 2 Web Ontology Language: Document Overview, W3C Recommendation, 27 October 2009.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, The Description Logic Handbook: Theory, Implementation, and Applications, 2nd Edition, Cambridge University Press, 2007.

[3] Y. Kazakov, $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$, in: Proc. 11th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'08), AAAI Press, 2008, pp. 274–284.

[4] M. Krötzsch, F. Maier, A. Krisnadhi, P. Hitzler, A better uncle for OWL: nominal schemas for integrating rules and ontologies, in: Proc. 20th Int. Conf. on World Wide Web (WWW'11), ACM, 2011, pp. 645–654.

[5] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. N. Grosof, M. Dean, SWRL: A Semantic Web Rule Language, W3C Member Submission, 21 May 2004, available at `http://www.w3.org/Submission/SWRL/`.

[6] T. Liebig, M. Luther, O. Noppens, M. Wessel, OWLlink, Semantic Web – Interoperability, Usability, Applicability 2 (1) (2011) 23–32.

[7] I. Horrocks, O. Kutz, U. Sattler, The even more irresistible $\mathcal{SROIQ}$, in: Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06), AAAI Press, 2006, pp. 57–67.

[8] F. Baader, S. Brandt, C. Lutz, Pushing the $\mathcal{EL}$ envelope, in: Proc. 19th Int. Joint Conf. on Artificial Intelligence (IJCAI'05), Morgan Kaufmann Publishers, 2005, pp. 364–369.

[9] Y. Kazakov, Consequence-driven reasoning for horn $\mathcal{SHIQ}$ ontologies, in: Proc. 21st Int. Conf. on Artificial Intelligence (IJCAI'09), Morgan Kaufmann Publishers, 2009, pp. 2040–2045.

[10] A. Steigmiller, B. Glimm, T. Liebig, Nominal schema absorption, in: Proc. 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI'13), AAAI Press, 2013, pp. 1104–1110.

[11] A. K. Hudek, G. E. Weddell, Binary absorption in tableaux-based reasoning for description logics, in: Proc. 19th Int. Workshop on Description Logics (DL'06), Vol. 189, CEUR, 2006.

[12] D. Tsarkov, I. Horrocks, Efficient reasoning with range and domain constraints, in: Proc. 17th Int. Workshop on Description Logics (DL'04), Vol. 104, CEUR, 2004.

[13] E. Sirin, From wine to water: Optimizing description logic reasoning for nominals, in: Proc. 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'06), AAAI Press, 2006, pp. 90–99.

[14] A. Steigmiller, T. Liebig, B. Glimm, Extended caching, backjumping and merging for expressive description logics, in: Proc. 6th Int. Joint Conf. on Automated Reasoning (IJCAR'12), Vol. 7364 of LNCS, Springer, 2012, pp. 514–529.

[15] A. Steigmiller, B. Glimm, T. Liebig, Coupling tableau algorithms for expressive description logics with completion-based saturation procedures, in: Proc. 7th Int. Joint Conf. on Automated Reasoning (IJCAR'14), LNCS, Springer, 2014, accepted.

[16] B. Glimm, I. Horrocks, B. Motik, R. Shearer, G. Stoilos, A novel approach to ontology classification, J. of Web Semantics: Science, Services and Agents on the World Wide Web 14 (2012) 84–101.

[17] V. Haarslev, R. Möller, A.-Y. Turhan, Exploiting pseudo models for tbox and abox reasoning in expressive description logics, in: Proc. 1st Int. Joint Conf. on Automated Reasoning (IJCAR'01), Vol. 2083 of LNCS, Springer, 2001, pp. 61–75.

[18] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: System description, in: Proc. 3rd Int. Joint Conf. on Automated Reasoning (IJCAR'06), Vol. 4130 of LNCS, Springer, 2006, pp. 292–297.

[19] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, Z. Wang, Hermit: An OWL 2 reasoner, J. of Automated Reasoning (2014) 1–25.

[20] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, J. of Web Semantics 5 (2) (2007) 51–53.

[21] Y. Kazakov, M. Krötzsch, F. Simančík, The incredible ELK, J. of Automated Reasoning (2013) 1–61.

[22] T. Gardiner, I. Horrocks, D. Tsarkov, Automated benchmarking of description logic reasoners, in: Proc. 19th Int. Workshop on Description Logics (DL'06), Vol. 198, CEUR, 2006.

[23] B. Smith, M. Ashburner, C. Rosse, J. Bard, W. Bug, W. Ceusters, L. J. Goldberg, K. Eilbeck, A. Ireland, C. J. Mungall, T. O. Consortium, N. Leontis, P. Rocca-Serra, A. Ruttenberg, S.-A. Sansone, R. H. Scheuermann, N. Shah, P. L. Whetzeland, S. Lewis, The OBO Foundry: coordinated evolution of ontologies to support biomedical data integration, Nature Biotechnology 25 (2007) 1251–1255.

[24] N. Matentzoglu, S. Bail, B. Parsia, A corpus of OWL DL ontologies, in: Proc. 26th Int. Workshop on Description Logics (DL'13), Vol. 1014, CEUR, 2013.

[25] V. Haarslev, R. Sebastiani, M. Vescovi, Automated reasoning in $\mathcal{ALCQ}$ via SMT, in: Proc. 23rd Int. Conf. on Automated Deduction (CADE'11), Springer, 2011, pp. 283–298.

[26] R. S. Gonçalves, S. Bail, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, B. Glimm, Y. Kazakov, OWL reasoner evaluation (ORE) workshop 2013 results: Short report, in: Proc. 2nd Int. Workshop on OWL Reasoner Evaluation (ORE'13), Vol. 1015, CEUR, 2013, pp. 1–18.