# A Reasoning Broker Framework for OWL

Jürgen Bock, Tuvshintur Tserendorj, Yongchun Xu, Jens Wissmann, and Stephan Grimm

FZI Research Center for Information Technology, Karlsruhe, Germany
{bock,tserendo,xu,wissmann,grimm}@fzi.de

**Abstract.** Semantic applications that utilise OWL ontologies can benefit from a broad range of OWL reasoning systems, which allow for the inference of implicit knowledge from explicitly given facts and axioms. Different OWL reasoners, however, specialise in different reasoning problems or kinds of ontologies, and hence perform differently in certain reasoning scenarios. This paper presents a reasoning broker framework, which connects to different existing reasoning systems and intelligently delegates reasoning requests. The behaviour of the broker is controlled by exchangeable and configurable broker strategies featuring selection and parallelisation of reasoners, centralised caching, simulated anytime reasoning, and various other potential features. A first experiment shows performance improvement for a sequence of queries compared to the use of different single reasoners.

## 1 Introduction

The Web Ontology Language (OWL) is to large extents based on Description Logics [1], a family of decidable fragments of first order logic. This characteristic allows OWL based semantic applications to perform automated reasoning in order to infer implicit knowledge from explicitly stated facts and axioms. In such applications different reasoning tasks arise, such as satisfiability checking, subsumption checking, classification, or instance retrieval. There can be various other derived reasoning tasks, such as the ones provided by the OWL API's [2] reasoner interface. Although theoretically all reasoning tasks can be reduced to satisfiability checking, different reasoning systems are optimised for different kinds of tasks. Furthermore different reasoning systems are optimised for different kinds of ontologies, such as ontologies with large ABoxes or complex TBoxes, and hence different reasoners perform differently in different scenarios [3–7]. The upcoming W3C standard for OWL 2 (Web Ontology Language version 2) [8] distinguishes between different Profiles [9], namely OWL 2 EL, OWL 2 QL, and OWL 2 RL, which trade language expressivity for reduced reasoning complexity. Currently there is a number of reasoning systems being developed focusing on efficient reasoning on those fragments. Apart from traditional sound and complete reasoning systems, such as Pellet [10], FaCT++ [11], or KAON2 [6], there are approximate reasoners, such as SCREECH [12] and AQA [13], which trade soundness and/or completeness for runtime performance. In certain scenarios where speed is highly desired, such a tradeoff can be tolerated.

## 2 Reasoning Brokerage

The idea behind reasoning brokerage is to utilise different existing reasoners in the background while providing a single interface to the user or application. The broker invokes connected remote reasoners in an intelligent manner such that their strengths and weaknesses are considered, and the shortest possible run-time for a particular reasoning task (or a sequence of tasks) is achieved. Refer to Sect. 3 for an overview on which presented features are currently realised.

**Parallel Reasoner Invocation.** Predicting the exact run-time of reasoning systems for a given ontology and a given query is hardly possible in practice. Even prediction of the first reasoner to finish a given query is not possible in many cases, which motivates the parallel execution of reasoning tasks on a set of reasoning systems. The reasoner that finishes first can then propagate the results of the query, assuming correctness of all reasoners invoked.

**Reasoner Selection.** Due to benchmarks and knowledge about the implementations of different reasoning systems, some of the available reasoners can be selected prior to actually executing the reasoning tasks. This selection will keep reasoners unsuitable for a given ontology/query combination idle and available for other reasoning tasks they are more appropriate for.

**Query Decomposition.** Queries containing complex class or property expressions can be analysed if those expressions decompose into several subexpressions which can be answered by different reasoners in parallel. It must be ensured, though, that the combination of the results delivered by different reasoners does not bear an unacceptable overhead compared to the performance gained by parallel computation. A naive example of such a decomposition is to split a conjunctive class expression into its operands and answer each subexpression in parallel. The answer in this simple case would be the intersection of the answers delivered by the different reasoners.

**Partitioning of Ontologies.** The notions of *conservative extensions* and *locality* provide means to partition an ontology into several semantically independent modules [14]. Executing a query on such a module instead of the whole ontology can result in run-time performance improvements for reasoning requests. Moreover, in combination with intelligent query decomposition more complex queries can be executed by different reasoners on different ontology modules in parallel.

**Load Balancing and Scheduling.** In a scenario where a sequence of queries is to be answered, or multiple applications are using the same instantiation of a reasoning broker, it is necessary to balance the workload of each remote reasoner in order to provide optimal overall run-time performance. To this end

an asynchronous reasoner interface would allow for acceptance of more than a single query at once from an application. Query answering can then be scheduled to be processed by the different remote reasoners according to their strengths and language conformance, in order to ensure maximum throughput of queries.

**Anytime Reasoning.** Anytime algorithms are designed to gradually improve the quality or quantity of their results as computation time increases, and end with providing the whole answer if complete computation is required. Anytime algorithms have been developed for the reasoning broker framework by combining approximate reasoning systems [12, 13]. The anytime behaviour can be achieved by either combining only the approximate methods or by combining them with a sound and complete reasoner.

**Real-time Benchmarking.** Assuming that the reasoning systems have been correctly implemented, existing benchmarks for ontology reasoning basically focus on performance measurements. It is also important to perform correctness tests for the implementation of the reasoning systems [7], in particular for the evaluation of emerging approximate reasoning systems with measuring the quality of answers as a special case of correctness tests. The reasoning broker provides an ideal infrastructure for both performance and correctness tests.

## 3 Implementation

The reasoning broker has been implemented as the HERAKLES system[1] in the Java™ programming language, based on the OWL API[2] [2]. HERAKLES is implemented in a client/server architecture to ensure modular decoupling of remote reasoners, *i.e.* the HERAKLES server, and the broker layer, *i.e.* the HERAKLES client. The HERAKLES client implements the `OWLReasoner` interface of the OWL API and can thus be used like any standard reasoner from within an OWL API based application. The HERAKLES client furthermore maintains a reasoner registry to record attached remote reasoners that can be used by the broker. Remote reasoners are wrapped into a remote reasoner adapter, which allows them to be run as reasoning servers connected to the HERAKLES client. This adaptation has not only been realised for OWL API compliant reasoners, but also for the KAON2 reasoner with its own API, and the KAON2 based approximate reasoning systems SCREECH and AQA. The communication between client and servers has been realised using Java™ RMI[3]. Hence reasoning servers can be run on remote machines, which allows for exclusive provision of computational resources for each reasoner.

In addition to the implementation of the OWL API reasoner interface, there is a plug-in available for the Protégé 4 ontology editor [15]. Via this plug-in

[1] `http://herakles.sourceforge.net`
[2] `http://owlapi.sourceforge.net`
[3] Remote Method Invocation

HERAKLES can be selected and used the same way as any standard reasoner from within Protégé 4. The plug-in provides additional functionality, namely *(i)* selection of remote reasoners to be used, *(ii)* strategy selection and configuration, *(iii)* anytime querying with asynchronous result delivery, and *(vi)* real-time statistics of run-time performance of the attached remote reasoners.

### 3.1 Broker Strategies

The behaviour of the broker and thus the implementation of the features discussed in Sect. 2 is controlled by exchangeable broker *strategies*. More precisely there is a load strategy to control the loading of ontologies into the different remote reasoners, and an execution strategy to control the execution of reasoning tasks by those reasoners. The strategy concept allows for easy substitution of both load and execution strategy by different implementations depending on the usage scenario of the reasoning broker. Furthermore the strategy concept allows for the implementation and use of customised strategies for specific use cases. Implementation of strategies in HERAKLES is simplified by several *strategy components*, which encapsulate core broker tasks such as parallelisation, reasoner selection, partitioning, or ontology analysing. These strategy components can then be used and combined to assemble new broker strategies. The following paragraphs describe interfaces and currently available implementations of strategy components and strategies for HERAKLES.

*Paralleliser.* This component invokes the execution of a reasoning task on a selection of reasoners in parallel. It will most likely be the final component involved in a strategy, possibly after some partitioning and selection steps. Currently there are two implementations: a competing paralleliser, which delivers the result of the reasoner that finishes first, and a blocking paralleliser, which waits until all reasoners have finished. The former will most likely be the default implementation in order to gain the best run-time performance of the broker, while the latter could be used for benchmarking tasks.

*Selector.* This component selects a set of reasoners out of the ones registered by the broker. Different implementations can apply different selection criteria, such as ontology properties, reasoning task to be executed, or query properties[4]. Currently there are two implementations: an ontology selector, which selects reasoners according to properties of the ontology, and a task selector, which selects reasoners according to the reasoning task to be performed. Selection of reasoners in this way requires knowledge about the capabilities of the different reasoners, which are currently recorded by the remote reasoner adapters.

*Modulariser.* This component provides means to partition an ontology into several modules, which can ideally be processed by different reasoners concurrently. There is currently no implementation available, but there are plans for realising partitioning as discussed in Sect. 2.

[4] Query properties could be for instance the language features used in a class description in an instance query.

*Analyser.* This component is supposed to be used in load strategies which perform an analysis of the ontologies to be loaded. The information gained by this analysis, *i.e.* characteristics of the ontologies, can then be used *e.g.* by selectors in the execution phase.

*Basic/Analysing Load Strategy.* This load strategy loads the ontology into all available remote reasoners. The *analysing load strategy* extends the basic load strategy by additionally analysing the ontologies and recording their characteristics.

*Basic/Fault-tolerant Parallelisation Strategy.* This execution strategy performs a reasoning request on all available (idling) remote reasoners, which have loaded the ontologies. A *fault-tolerant parallelisation strategy* extends the basic parallelisation strategy by being insensitive to failing remote reasoners. In case of a failure, it waits for more reasoners to become available and fails on a particular query only if all remote reasoners fail.

*(Fault-tolerant) Task Selection Strategy.* This execution strategy selects remote reasoners according to the reasoning task requested. Selection is carried out by a *task selector* strategy component, which can be configured in order to map reasoning tasks to reasoners having certain characteristics. Selected reasoners are then invoked in parallel using the *competing paralleliser* strategy component. A *fault-tolerant task selection strategy* extends the task selection strategy by being insensitive to failing remote reasoners. In the case all selected reasoners fail on a particular query, it also selects reasoners not matching the selection criteria in order to try and have the query succeed[5].

*Anytime Strategy.* This execution strategy simulates anytime reasoning behaviour by using approximate reasoning systems as discussed in Sect. 2. It selects distinct sets of remote reasoners respecting soundness, completeness, and both soundness and completeness. All reasoners are invoked in parallel using the *competing paralleliser* strategy component, where each set of reasoners is invoked by a different paralleliser. Results are delivered from the fastest reasoner of each set, characterising results as *sound*, *complete*, or *sound/complete* rsp. Anytime behaviour arises from the faster run-times of the approximate reasoners and thus from the early delivery of (potentially) unsound or incomplete answers.

*Benchmark Strategy.* This execution strategy can be used for simple run-time performance benchmarking of reasoners. It invokes all available remote reasoners in parallel without any prior selection. The strategy component for parallel execution is the *blocking paralleliser* to enable time measurement of each reasoner for each reasoning task. The blocking characteristic of this strategy component ensures availability of all reasoners for each reasoning task out of a test series.

[5] This behaviour assumes that the selection in the first place was only based on expected run-time performance and not due to language conformance.

### 3.2 Query Stream Experiment

To demonstrate the effect of the reasoning broker using multiple remote reasoners compared to the use of single reasoners, an experiment has been conducted to simulate the behaviour of answering a sequence of queries on a single ontology. Such a setting, in which the ontology or set of ontologies is rather stable, are typical for most domain specific semantic applications[6].

Note that this experiment is *not* intended to be yet another benchmark of the reasoners used. There was neither a comparison of the results delivered by the different reasoners, nor of the time needed by each reasoner for a single query.

**Setup.** In this experiment a sequence of queries is asked on a modified version of the Wine ontology. It is the same version as used in previous benchmarks [6, 4], enriched by a datatype property in order to be able to ask queries containing some datatype expression. A set of 100 queries was randomly generated using all reasoning query methods of the OWL API, and arbitrary class and property expressions using named entities of the ontology. The length of the query expressions was limited to use at most 6 named entities from the ontology. It is not relevant for the purpose of this experiment, that queries might be meaningless since results are not compared and inference of an empty result is itself challenging. Note that there was also no explicit request (query) for classification or realisation of the ontology. The reasoners used as remote reasoner in this experiment are Pellet 2.0.0-RC7, FaCT++ 1.3.0, and KAON2 (2008-06-29).

The experiment was conducted in a distributed computing environment using four identical Linux (2.6.18-8) machines with four 1.86 GHz Intel® Xeon® CPUs, and 1 GB physical memory each. Each reasoner and the HERAKLES client were running on different machines to guarantee exclusive use of computing resources for each reasoner. HERAKLES was set up with the *basic load strategy* and *fault tolerant parallelisation strategy.*

**Result.** It could be observed that Pellet was the only reasoner to answer all queries without failing. KAON2 is unable to answer requests, where the class description contains nominal expressions, which was the case in 10 test queries. FaCT++ does not support a number of OWL API reasoner methods using its OWL API wrapper implementation. For this reason, it could not handle 32 test queries. It should be noted, that some of the queries that could only be answered by Pellet were rather complex, which explains the conspicuously slower runtime of Pellet in this experiment. In HERAKLES a fault tolerant parallelisation strategy was used (see Sect. 3.1), which made it recover from a failing reasoner by waiting for other reasoners to become available in order to answer these queries. As Fig. 1 shows, the simultaneous use of all three reasoners resulted in a significant improvement of run-time performance compared to solely using Pellet, while retaining faultless processing of the query stream.

[6] It is assumed that such a semantic application uses the OWL API as ontology management back-end interface.

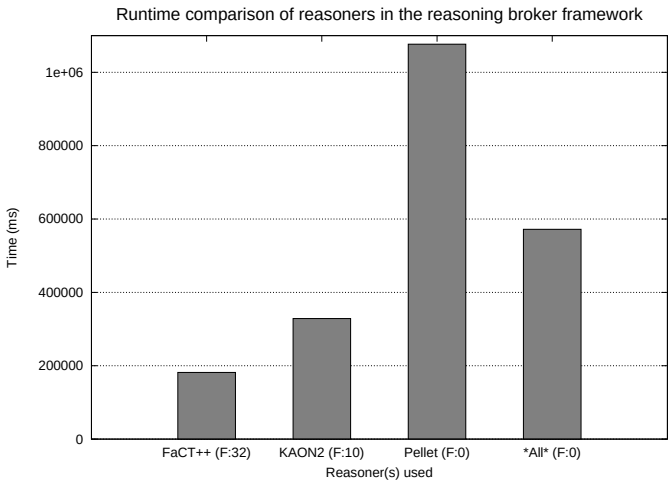


**Fig. 1.** Comparison run-time performances of HERAKLES used with different remote reasoners for a sequence of 100 queries. (The numbers in parentheses indicate the number of failures, *i.e.* queries that could not be answered.)

## 4 Conclusion

The variety of language features available in the upcoming W3C standard for OWL 2 and its profiles, as well as the multitude of available OWL reasoners makes it a challenge to choose the best performing reasoner on a given ontology and reasoning task. This problem is tackled by a reasoning broker framework, which connects to different standard reasoners and provides a number of features in order to increase overall run-time performance for reasoning tasks. Among those features are selection and parallel invocation of reasoners, ontology partitioning, query decomposition, anytime reasoning, *etc.* Some of these features have already been implemented in the HERAKLES system. A first experiment asking a sequence of queries on a static ontology shows HERAKLES to perform favourable using several reasoners compared to the use of a single one.

HERAKLES is in active development, and all features as pointed out in Sect. 2 will be implemented soon. Most prominently the focus will be on ontology partitioning and query decomposition. There are also plans to include reasoner selection heuristics in order to select suitable reasoners based on their expected run-time behaviour. To this end, machine learning techniques will be applied in order to classify reasoners according to their performance on ontologies and queries with certain properties. There will also be efforts towards adopting the OWLLink[7] protocol both for the interface to HERAKLES, as well as for the communication between the HERAKLES client and the remote reasoning servers.

[7] `http://www.owllink.org/`

## Acknowledgement

The presented research was funded by the German Federal Ministry of Economics (BMWi) under the project Theseus (number 01MQ07019).

## References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook – Theory, Implementation and Applications. Cambridge University Press (2003)
2. Horridge, M., Bechhofer, S., Noppens, O.: Igniting the OWL 1.1 Touch Paper: The OWL API. In: OWLED. Volume 258 of CEUR Workshop Proceedings., CEUR-WS.org (2007)
3. Luther, M., Liebig, T., Böhm, S., Noppens, O.: Who the Heck is the Father of Bob? – A Survey of the OWL Reasoning Infrastructure for Expressive Real-World Applications. In: Proc. of the 6th European Semantic Web Conference (ESWC). Volume 5554 of LNCS., Berlin, Springer (May 2009) 66–80
4. Bock, J., Haase, P., Ji, Q., Volz, R.: Benchmarking OWL Reasoners. In: Proc. of the AReа2008 Workshop, Tenerife, Spain (June 2008)
5. Weithöner, T., Liebig, T., Luther, M., Böhm, S., von Henke, F.W., Noppens, O.: Real-World Reasoning with OWL. In: Proc. of the 4th European Semantic Web Conference (ESWC). Volume 4519 of LNCS., Berlin, Springer (June 2007) 296–310
6. Motik, B.: Reasoning in Description Logics using Resolution and Deductive Databases. PhD thesis, Universität Karlsruhe (2006)
7. Gardiner, T., Tsarkov, D., Horrocks, I.: Framework For an Automated Comparison of Description Logic Reasoners. In: Proc. of the 5th Int. Semantic Web Conf. (ISWC). Volume 4273 of LNCS., Berlin, Springer (2006) 654–667
8. Golbreich, C., Wallace, E.K.: OWL 2 Web Ontology Language: New Features and Rationale. W3C working draft, W3C (June 2009) `http://www.w3.org/TR/2009/WD-owl2-new-features-20090611/`.
9. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: OWL 2 Web Ontology Language:Profiles. W3C candidate recommendation, W3C (June 2009) `http://www.w3.org/TR/2009/CR-owl2-profiles-20090611/`.
10. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics **5**(2) (June 2007) 51–53
11. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR). Volume 4130 of LNAI., Berlin, Springer (August 2006) 292–297
12. Tserendorj, T., Rudolph, S., Krötzsch, M., Hitzler, P.: Approximate OWL-Reasoning with Screech. In: Proc. of the 2nd Int. Conf. on Web Reasoning and Rule Systems. Volume 5341 of LNCS., Berlin, Springer (October 2008) 165–180
13. Tserendorj, T., Grimm, S., Hitzler, P.: Approximate Instance Retrieval. Technical report, FZI Research Center for Information Technology, Karlsruhe, Germany (December 2008) availavle at `http://www.aifb.uni-karlsruhe.de/WBS/phi/resources/publications/approxInstRetr08.pdf`.
14. Grau, B.C., Horrocks, I., Kazakov, Y., Sattler, U.: Just the Right Amount: Extracting Modules from Ontologies. In: Proc. of the 16th Int. Conf. on World Wide Web (WWW), New York, NY, USA, ACM (May 2007) 717–726
15. Bock, J., Tserendorj, T., Xu, Y., Wissmann, J., Grimm, S.: A Reasoning Broker Framework for Protégé. 11th Int. Protégé Conf., Amsterdam (June 2009)