# Reasoning Brokerage: New Reasoning Strategies

**Jürgen Bock**

**Abstract** Once a formal representation of data is available an important issue is to infer additional aspects out of this knowledge base. But according to the type of representation scheme chosen, different techniques can be applied or gain better or more accurate results. A reasoning broker system offers the possibility to apply strategies for selecting the best reasoning system, or for letting run different reasoners in parallel. In this article a reasoning broker system enabling the usage and integration of remote reasoners is presented. Additionally, the new reasoning capability of anytime reasoning has been developed and integrated into the reasoning broker.

## 1 Introduction

One of the key advantages of representing domain knowledge in ontologies is the formal semantics provided by the ontology language's logical underpinning. For instance, the Web Ontology Language (OWL) (Bock et al. 2009a) is based on description logics (Baader et al. 2003), a family of decidable fragments of first-order logic. This formal semantics enables the possibility of conducting logical reasoning in order to infer implicit knowledge from explicitly stated axioms and facts. In order to gain this added value, powerful inference engines (reasoners) are required, which draw conclusions on the provided ontology.

Depending on the use case at hand, the extent to which logically expressive language features are exploited in ontologies differs. Studies show that, in fact, most of the ontologies found on the Web are of low expressiveness in terms of language features used (Bock et al. 2008). Other differences that can be observed are in the

J. Bock (✉)
FZI, Forschungszentrum Informatik, Karlsruhe, Germany
e-mail: bock@fzi.de

size of ontologies with respect to the number of entities referenced by the ontology's axioms. These sizes can range from under a dozen up to several tens of thousands of entities. A significant characteristic when analyzing the size of an ontology is the type of the entities that occur in large numbers. For instance, there are ontologies that contain a large number of classes, thus forming a taxonomically rich knowledge model. On the other hand, there are ontologies with a smaller number of classes and statements about their terminological relations, but a great amount of instances asserted to those classes.

This variety of characteristics that discriminate the ontology landscape impose different challenges on a reasoning engine, which consequently has to cope with these ontologies. In the past decade, a plethora of reasoners has been developed for OWL.[1] Some reasoners, such as HermiT[2] or Pellet[3] can serve as reference implementations for the Description Logic fragment underlying OWL and DL, and thus are capable of processing ontologies that exploit the full logical expressiveness. Other reasoners, such as KAON2,[4] strive for efficient reasoning with ontologies containing a great number of instances, at the cost of slightly reduced logical expressiveness. With the standardization of OWL 2, a range of language profiles (Calvanese et al. 2009) was introduced, which deliberately reduce the expressive power in order to reduce reasoning complexity. Reasoners such as CEL,[5] TrOWL,[6] or ELLY[7] implement such language profiles. Table 1 shows a (non-exhaustive) list of reasoners together with some of their properties.

Apart from the obvious differences in language expressiveness reflected by OWL profiles, there are more subtle characteristics, for instance in the combination of language features, which have significant performance impact on the reasoning calculus. Developers of reasoners typically implement various optimization strategies that are automatically switched on, if the input ontology allows this. Nevertheless, there are major differences in the runtime performance of state-of-the-art reasoning systems depending on the input ontology and the reasoning task at hand. The various strengths and weaknesses of reasoners have been studied extensively in recent years (Bock et al. 2008; Gardiner et al. 2006; Liebig 2006; Luther et al. 2009; Pan 2005), leading to the conclusion that there is no single best reasoning system for all reasoning scenarios. Moreover, this leads to a major inconvenience for the developer of any semantic application intending to utilize reasoning capabilities, since choosing the best reasoner is a nontrivial task. In particular, in the case where either input ontologies or reasoning tasks change, there might be different reasoners suitable for different invocations.

[1] http://www.w3.org/2007/OWL/wiki/Implementations

[2] http://www.hermit-reasoner.com/

[3] http://clarkparsia.com/pellet/

[4] http://kaon2.semanticweb.org/

[5] http://lat.inf.tu-dresden.de/systems/cel/

[6] http://trowl.eu/

[7] http://elly.sourceforge.net/

**Table 1** Non-exhaustive selection of state-of-the-art reasoning systems (Source: Bock et al. (2012))

| | | |
|---|---|---|
| CEL | Native Profiles | EL |
| | Semantics | Direct |
| | (Non-)conformance | Lacks support for nominals (ObjectHasValue and ObjectOneOf) and data types/values |
| | Algorithm | Proprietary |
| | API | OWL API (not for v3.1) |
| | Authorization | Open source |
| ELLY | Native profiles | EL, RL |
| | Semantics | Direct |
| | (Non-)conformance | OWL profile support under development |
| | Algorithm | Rule inferencing |
| | API | OWL API (old version) |
| | Authorization | Open source |
| FaCT++ | Native profiles | DL |
| | Semantics | Direct |
| | (Non-)conformance | Fully conforming except for keys and some data types |
| | Algorithm | Tableau |
| | API | OWL API, DIG interface |
| | Authorization | Open source |
| HermiT | Native profiles | DL |
| | Semantics | Direct |
| | (Non-)conformance | Fully conforming |
| | Algorithm | Hypertableau |
| | API | OWL API |
| | Authorization | Open source |
| Pellet | Native profiles | DL, EL |
| | Semantics | Direct |
| | (Non-)conformance | Fully conforming |
| | Algorithm | Tableau |
| | API | OWL API, Jena API, DIG interface |
| | Authorization | Open source |
| RacerPro | Native profiles | DL |
| | Semantics | Direct |
| | (Non-)conformance | No nominals and RBox |
| | Algorithm | Tableau |
| | API | OWL API, DIG interface, OWLlink |
| | Authorization | Commercial (free for research) |
| TrOWL | Native profiles | DL, EL, QL |
| | Semantics | Direct |
| | (Non-)conformance | |
| | Algorithm | Various proprietary |
| | API | OWL API (not for v3.1) |
| | Authorization | Open-source |

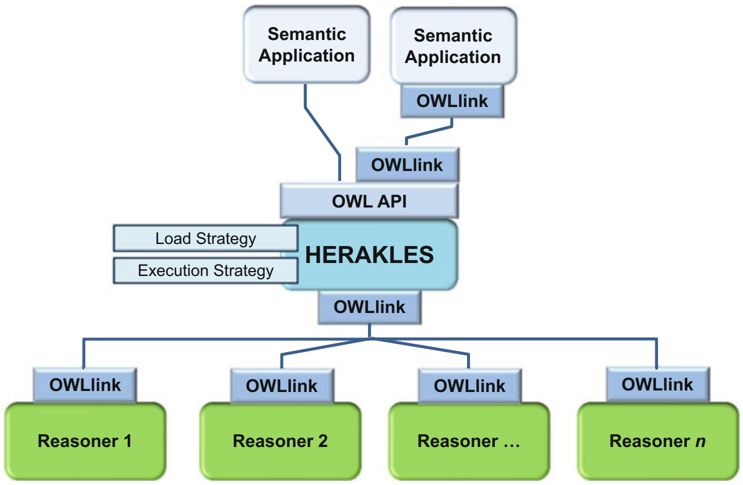

**Fig. 1** Architecture of the HERAKLES reasoning broker (Source: Bock et al. (2012))

This article describes how the problem can be addressed using a reasoning broker approach, which serves as a hub between existing reasoners and a client. For the client, this broker appears as a single reasoner but delegates reasoning request intelligently to its (specialized) reasoning engines in the back end. The broker thus utilizes the capabilities of all connected reasoners, and provides additional added value, such as anytime reasoning simulation by approximation.

In the following Sect. 2, the general architecture of the reasoning broker system is introduced, which can be configured by several broker strategies, which are presented in Sect. 3. The special case of anytime behavior by approximation is dealt with in Sect. 4, before the article concludes in Sect. 5.

## 2 Broker Architecture

A reasoning broker framework for OWL has been implemented as the HERAKLES system[8] (Bock et al. 2009b,c) in the context of the THESEUS research program. The overall architecture of the system is illustrated in Fig. 1.

From an abstract point of view, the broker connects to several external 3rd party reasoning systems in the back end, while serving as a single reasoner endpoint for a semantic client application. The way the broker interacts with the external reasoning systems is driven by two broker strategies: A load strategy and an execution strategy, each of which is exchangeable in order to accommodate the

[8]http://herakles.sourceforge.net

particular usage scenario of the broker (see Sect. 3). Moreover, since the broker is the central component of a distributed reasoning infrastructure, it can implement centralized caching and load balancing mechanisms, in order to provide instant response for reoccurring requests, and efficiently manage the available reasoner resources.

From a more technical point of view, the HERAKLES system is internally based on the OWL API (Horridge and Bechhofer 2011), a popular, Java-based framework for dealing with OWL ontologies and a number of conforming reasoners. Due to this, semantic applications that are based on the OWL API as well can directly use the reasoner interface implementation provided by HERAKLES. On the other hand, HERAKLES implements the OWLlink protocol (Liebig et al. 2010), a standardization effort aiming at setting up a common interface in order to connect different semantic applications in particular reasoners. Adapting to this, HERAKLES can connect with arbitrary OWL reasoners that exhibit themselves as OWLlink servers.[9] Since the OWLlink protocol is HTTP-based, HERAKLES can operate with remote reasoners in a distributed environment. The fact that HERAKLES itself acts as an OWLlink server enables the setup of an intelligent reasoning service in a highly distributed environment.

## 3 Broker Strategies

The reasoning broker does not implement a reasoning engine or an inference calculus of any kind. Instead, reasoning requests are intelligently delegated to external third party reasoners that might be specialized for efficient processing of particular requests. This intelligent delegation is controlled by two broker strategies that are integrated into the broker as independent modules:

- A *load strategy* is responsible for loading ontologies into the registered external reasoners.
- An *execution strategy* is responsible for invoking the external reasoners for particular reasoning requests.

Both, load and execution strategy are exchangeable in order to accommodate the particular broker usage scenario at hand.

**Loading** The load strategy manages the initialization of the broker and all (relevant) registered external reasoners with the ontologies in the current reasoning scenario. Different load strategies are possible. A *basic* strategy simply asks all registered reasoners to load the ontologies as requested by the client. An *analyzing* strategy extracts various features from the ontologies in order to use this information for intelligent selection of reasoners for particular reasoning tasks (see Sect. 3).

[9]Reasoners that do not provide an OWLlink server interface have to be wrapped with minor effort.

**Parallelization** The scalability issue in the context of semantic technologies becomes more and more important. One way to address scalability in terms of ontology reasoning is to utilize parallelization (Bock 2008) techniques where possible. A straightforward way of using a reasoning broker for executing reasoning requests is to delegate the request to all registered (and idling) reasoners and return the result of the reasoner that answers first. However, it has to be ensured that the reasoners do not consume shared computing resources which would slow down the algorithms due to reduced computational power. Thus the best setting for utilizing such a strategy would be a distributed computing infrastructure with independent compute nodes, or at least guaranteed computational resources for each node (e.g. in a virtual environment, such as compute clouds).

**Selection** In case distributed computing resources are rare, or there is any other reason to save computational resources, it is advisable to avoid blind invocation of reasoning tasks on all available reasoners. Thus, using the features extracted by an analyzing load strategy, a *selection* strategy can determine the best suitable reasoner(s) for a given request. Apart from looking at simple rules based on the known capabilities of the various reasoners, a machine learning approach has been developed (Bock et al. 2012). In this approach, a corpus of real-world ontologies from the Web and a set of generated queries are used to determine the fastest reasoner for each ontology/query combination. The features of ontology and query are used to teach a model that is later used to predict the best suitable reasoner for any new ontology/query combination. Experiments have shown that this selection strategy can predict the best suitable reasoner with an accuracy of up to 77 %.

**Fault Recovery** Parallelization and the availability of a number of different reasoners can also be exploited to improve the robustness of the whole reasoning process. A *fault tolerant strategy* has been developed, which works similarly to the basic parallelization strategy, but reacts to failures of external reasoners. Such failures can occur if a reasoner does not support a particular reasoning request, or if a reasoner is in a prototypical and thus unstable state. Since a number of reasoners is invoked simultaneously, the failure of one reasoner does not cause the others to stop. An experiment with a series of subsequent queries was carried out in order to test the fault recovery achieved by the said strategy (Bock et al. 2009b). A set of 100 queries was executed using the HERAKLES reasoning broker. In four test runs HERAKLES was configured three times with a single reasoner registered (FaCT++, KAON2, and Pellet), and once with all reasoners registered. As Fig. 2 shows, the configuration with FaCT++ shows the best runtime performance. However, FaCT++ failed in 32 of the 100 queries. Compared to that, Pellet failed in no case, at the cost of higher runtime. (In fact it could be observed that the queries that caused long runtimes for Pellet are the ones that caused FaCT++ to fail rather quickly, which explains the runtime discrepancy.) Using the broker configuration with all reasoners registered and the fault tolerant strategy activated, all queries could be processed with no failures in less time than when using Pellet alone. The case where a series of queries is executed on a fixed set of ontologies is assumed

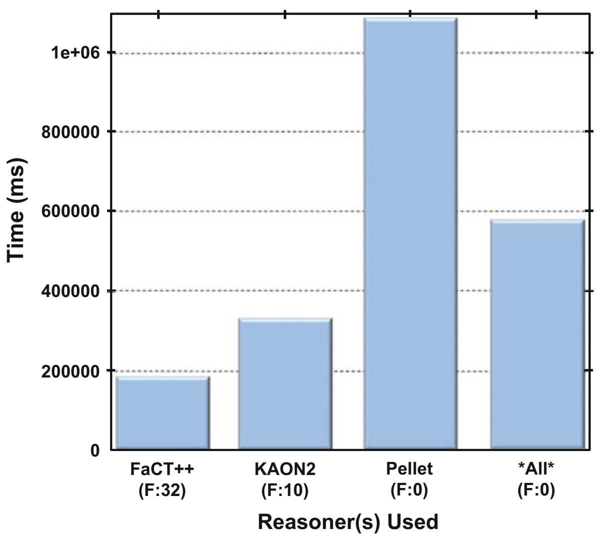

**Fig. 2** Runtime comparison of reasoners in the reasoning broker framework: Query stream experiment with 100 queries. The bars denote the total runtimes using the broker configured with a single reasoner, or with all reasoners (*right bar*). Numbers in parentheses denote the number of failures (Sources: Bock et al. (2008, 2009b))

to be a common one. Thus a broker framework such as HERAKLES provides some real added value in terms of fault tolerance and runtime improvement.

## 4 Anytime Reasoning Through Approximation

Most reasoners are developed with the goal of providing sound and complete answers to queries. For the reasoning task of instance retrieval, this would mean that a request to obtain instances of a class description answers with a set which on the one hand contains only *correct* instances (soundness), and on the other hand *all* instances (completeness) for which it can logically be inferred that they belong to the given class description. These soundness and completeness guarantees are causing the algorithms to invoke complex inference procedures of high computational complexity. Recent approaches developed in the context of the THESEUS research program give up these soundness and completeness guarantees for an improved runtime performance by approximation (Tserendorj 2010). The Screech system developed in THESEUS is based on the KAON2 reasoner[10] and comes in three variants (Tserendorj et al. 2008):

- Screech-all: This variant ensures completeness but gives up soundness.
- ScreechNone: This variant ensures soundness but gives up completeness.
- Screech-one: This variant gives up both soundness and completeness.

[10]http://kaon2.semanticweb.org/

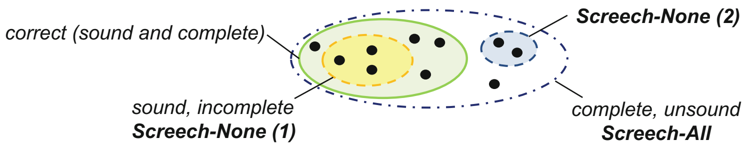

**Fig. 3** Combination of variants of the approximate reasoning system SCREECH in order to achieve simulated anytime behavior for instance retrieval requests

It has been analyzed, how the combination of different kinds of anytime algorithms exhibits an anytime behavior that cannot be achieved using a single algorithm (Rudolph et al. 2008). The SCREECH variants introduced above can be arranged in such a setting as illustrated in Fig. 3. For a given class expression, an instantiation of SCREECH-NONE is invoked, which is sound but incomplete, and thus delivers correct instances of this class expression, but the result set might not be complete. Invoked at the same time, an instantiation of SCREECH-ALL, which is complete but unsound, delivers a set that contains all correct instances, but potentially also incorrect ones. A second SCREECH-NONE instance invoked with the *complement* of the original class expression delivers a set of correct negative results, and thus can be used to determine a subset of the result set from SCREECH-ALL that is definitely incorrect. Though the SCREECH instances work as black boxes, they can be invoked in parallel and will probably show different runtimes. After any reasoner returns its results, they can be delivered to the client, provided the soundness and completeness properties are indicated accordingly. Thus, an example run may provide results as follows. First, SCREECH-ALL provides a set of instances which might contain incorrect ones. Second, SCREECH-NONE (positive query) returns with a subset of the results from SCREECH-ALL. These are definitely correct and can be marked accordingly. Third, SCREECH-NONE (negative query) returns with a subset of the results from SCREECH-ALL, which are definitely incorrect and can be removed from the result set. The final result is a better approximation than using any of the SCREECH variants alone. In order to finally get the correct (sound and complete) result set, a standard reasoner may have been invoked in parallel as well, which is expected to have a slower runtime performance, but is able to verify the final result. These steps in result delivery provide the client with approximate intermediate results that are subsequently refined.

An *anytime strategy* was implemented for the reasoning broker HERAKLES, which uses the broker facilities of parallelization. To this end, a novel reasoner interface was implemented to allow for asynchronous reasoning calls – a feature required for anytime result delivery. The novel interface and broker strategy were prototypically integrated into the ontology authoring tool Protégé (Bock et al. 2009c), where a color encoding was used to indicate the "certainty" of instances in the result set.

## 5 Conclusion

Reasoning brokerage was introduced as a concept that tackles the problem of diversity in reasoner performance, which can be observed for existing reasoning systems. To this end, the broker framework HERAKLES was presented as a solution for semantic application developers, who face the problem of selecting the most appropriate reasoning system for a given task. On the one hand, the broker is flexible in terms of exchangeable broker strategies, while on the other hand it simplifies the usability by providing the standardized programming interfaces of the OWL API and OWLlink to the application developer. A collection of broker strategies has been implemented accommodating parallelization, selection (including machine learning techniques), fault recovery, and anytime simulation through combinations of approximate reasoning systems.

Apart from facilitating the incorporation of existing reasoners in semantic applications, the reasoning broker system can encourage developers of reasoning systems to focus on particular optimizations for specific reasoning problems, without losing a large number of potential clients. HERAKLES is designed to ease the development of additional broker strategies, which might be required in a particular usage scenario. This flexibility supports the growing interest in semantic technologies on the level of powerful and suitable reasoning systems for intelligent applications.

## References

F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications* (Cambridge University Press, New York, 2003)

C. Bock, A. Achille Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Alan Ruttenberg, U. Sattler, M. Smith, OWL 2 Web Ontology Language – Structural Specification and Functional-Style syntax (2009a), http://www.w3.org/TR/owl2-syntax/

J. Bock, Parallel computation techniques for ontology reasoning, in *The Semantic Web – ISWC 2008*, ed. by A.P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T.W. Finin, K. Thirunarayan. Volume 5318 of Lecture Notes in Computer Science (Springer, Berlin/Heidelberg/New York, 2008), pp. 901–906, http://dblp.uni-trier.de/db/conf/semweb/iswc2008.html#Bock08

J. Bock, P. Haase, Q. Ji, R. Volz, Benchmarking OWL reasoners, in *Proceedings of the ARea2008 – Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*, Tenerife, June 2008, ed. by F. van Harmelen, A. Herzig, P. Hitzler, Z. Lin, R. Piskac, G. Qi. CEUR Workshop Proceedings, vol. 350, http://ceur-ws.org

J. Bock, U. Lösch, H. Wang, Automatic reasoner selection using machine learning, in *Proceedings of the 2nd International Conference on Web Intelligence, Mining and Semantics*, Helsingborg, ed. by D.D. Burdescu, R. Akerkar, C. Badica (ACM, New York, 2012), http://dblp.uni-trier.de/db/conf/wims/wims2012.html#BockLW12

J. Bock, T. Tserendorj, Y. Xu, J. Wissmann, S. Grimm, A reasoning broker framework for OWL, in *Proceedings of the 6th International Workshop on OWL: Experiences and Directions (OWLED*

*'09)*, Chantilly, ed. by R. Hoekstra, P.F. Patel-Schneider. CEUR Workshop Proceedings, vol. 529 (2009b), http://www.webont.org/owled/2009/papers/owled2009_submission_13.pdf

J. Bock, T. Tserendorj, Y. Xu, J. Wissmann, S. Grimm, A reasoning broker framework for protégé, in *11th International Protégé Conference*, Amsterdam (June 2009c)

D. Calvanese, J. Carroll, G. De Giacomo, I. Herman, B. Parsia, P.F. Patel-Schneider, A. Ruttenberg, U. Sattler, OWL 2 Web Ontology Language – Profiles (2009), http://www.w3.org/TR/owl2-profiles/

T. Gardiner, I. Horrocks, D. Tsarkov, Automated benchmarking of description logic reasoners, in *Proceedings of the 19th International Workshop on Description Logics*, ed. by B. Parsia, U. Sattler, D. Toman. Volume 189 of CEUR Workshop Proceedings (Windermere, Lake District, UK, 2006), http://dblp.uni-trier.de/db/conf/dlog/dlog2006.html#GardinerHT06

M. Horridge, S. Bechhofer, The OWL API: a Java API for OWL ontologies. Semant. Web **2**(1), 11–21 (2011), http://dblp.uni-trier.de/db/journals/semweb/semweb2.html#HorridgeB11

T. Liebig, Reasoning with OWL – system support and insights. Technical Report 2006-04, Universität Ulm - Fakultät für Informatik, Sept 2006

T. Liebig, M. Luther, O. Noppens, M. Wessel, OWLlink. Semant. Web – Interoper. Usability, Appl. **2**(1), 23–32 (2010)

M. Luther, T. Liebig, S. Böhm, O. Noppens, Who the Heck is the father of Bob? A survey of the OWL reasoning infrastructure for expressive real-world applications, in *The Semantic Web: Research and Applications*, ed. by L. Aroyo, P. Traverso, F. Ciravegna, P. Cimiano, T. Heath, E. Hyvönen, R. Mizoguchi, E. Oren, M. Sabou, E. Simperlm. Volume 5554 of Lecture Notes in Computer Science (Springer, Berlin/Heidelberg/New York, 2009), pp. 66–80

Z. Pan, Benchmarking DL reasoners using realistic ontologies, in *Proceedings of the OWL: Experiences and Directions Workshop*, Galway, ed. by B.C. Grau, I. Horrocks, B. Parsia, P.F. Patel-Schneider. Volume 188 of CEUR Workshop Proceedings (2005), http://dblp.uni-trier.de/db/conf/owled/owled2005.html#Pan05

S. Rudolph, T. Tserendorj, P. Hitzler, What is approximate reasoning? in *Web Reasoning and Rule Systems*. Volume 5341 of Lecture Notes in Computer Science (Springer, Berlin/Heidelberg/New York, 2008), pp. 150–164

T. Tserendorj, Approximate assertional reasoning over expressive ontologies. PhD thesis, Fakultät für Wirtschaftswissenschaften, Karlsruher Institut für Technologie (KIT), Jan 2010, http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1235410

T. Tserendorj, S. Rudolph, M. Krötzsch, P. Hitzler, Approximate OWL-reasoning with screech, in *Web Reasoning and Rule Systems*, ed. by D. Calvanese, G. Lausen. Volume 5341 of Lecture Notes in Computer Science (Springer, Berlin/Heidelberg/New York, 2008), pp. 165–180, http://dblp.uni-trier.de/db/conf/rr/rr2008.html#TserendorjRKH08