Who the Heck is the Father of Bob?
A Survey of the OWL Reasoning Infrastructure for Expressive Real-World Applications

Marko Luther[1], Thorsten Liebig[2], Sebastian Böhm[1], and Olaf Noppens[2]

[1] DOCOMO Communications Laboratory Europe GmbH, Munich, Germany
`lastname@docomolab-euro.com`
[2] Inst. of AI, Ulm University, Ulm, Germany
`firstname.lastname@uni-ulm.de`

Abstract. Finding the optimal selection of an OWL reasoner and service interface for a specific ontology-based application is challenging. Over time it has become more and more difficult to match application requirements with service offerings from available reasoning engines, in particular with recent optimizations for certain reasoning services and new reasoning algorithms for different fragments of OWL. This work is motivated by real-world experiences and reports about interesting findings in the course of developing an ontology-based application. Benchmarking outcomes of several reasoning engines are discussed – especially with respect to accompanying sound and completeness tests. We compare the performance of various service and communication protocols in different computing environments. Hereby, it becomes apparent that these largely underrated components may have an enormous impact on the overall performance.

1 Introduction

In the recent past, the application of Semantic Web technologies has seen a steady increase in a broad variety of services and applications. On the foundation of W3C's Web Ontology Language (OWL) [1], Semantic Technologies are meant to be soon adopted by the industry. In the course of developing applications that rely on ontology based reasoning mechanisms, not only an appropriate reasoning engine needs to be selected. In fact, finding the right combination of reasoner and communication interface for expressive ontologies with the required level of expressivity is crucial for the overall reasoning performance. Existing benchmarks promise to shed some light on the performance of popular inference engines, at least for some selected ontologies and thus particular reasoning tasks. Our own experience has shown that most of the benchmark results are not as helpful as desired. When building an ontology based application, designers typically have to deal with many different requirements and constraints on the query interface, ontology updates, reliability of results, etc. To utilize ontology reasoning for applications in the mobile space, we first tried to identify an appropriate reasoning infrastructure by applying standard benchmarks as well as spot tests for different ontologies. In a second step, concrete reasoning scenarios for our application with specifically designed ontologies have been evaluated. In this paper, our experiences in the development are discussed to help others in realizing an appropriate reasoning infrastructure.

The remainder of this work is organized as follows: Section 2 gives an overview of existing reasoning engines and discusses some benchmarking results of the former. In the subsequent Section 3, a set of tests gained from developing real-world applications are performed. Concluding remarks can be found in Section 4.

2 OWL Reasoning Infrastructure

The core element of any OWL infrastructure consists of a reasoning engine capable of processing OWL ontologies at a certain level of expressivity. The typical processing services cover consistency and entailment checks as well as any kind of query answering. Important criteria of these services are soundness resp. completeness and practical efficiency. It is worth noting that both criteria are of equal importance and inherently depend on each other. For instance, efficiency can easily be achieved by giving up sound- and completeness. On the other hand, practical efficiency in a real-world usage scenario is without doubt an important requirement for semantic applications, even if reasoning with OWL is known to be unfeasible in the worst case. Nevertheless, highly optimized implementations of well-chosen calculi have empirically proven that real-world applications are possible. Besides, the variety of profiles of OWL 2 provides a selection of language fragments with different characteristics for various needs and ontology sizes.

Finding the appropriate reasoning approach resp. system for a given expressivity and problem size is all but obvious. Certain approaches perform well only with specific language fragments and the performance of reasoning engines may vary significantly from case to case because of subtle optimization techniques. However, when it comes to large volumes of ontology data, incremental reasoning or answer caching may be more important than any other optimization technique. Besides all these partly conflicting requirements, an application designer needs to know whether or not the chosen reasoning engine is mature and reliable enough for a particular setting. In other words, if a system is not sound (for whatever reason), it is practically useless for the vast majority of applications.

This sections summarizes empirical results from currently available OWL reasoning systems with respect to the above mentioned issues. Our findings only draw a very fragmentary picture, even though valuable insights can still be given. Our survey is based on experiences with a benchmark suite for large ABox data on the one hand, and a selection of small but difficult T- and ABox test cases on the other hand. This work updates and extends previous analyses with respect to hard spot tests [2] as well as huge volumes of instance data [3].

Our evaluation shows that even with only one benchmark suite, it is hard to draw a reliable conclusion. The runtime of systems heavily varies from case to case and is utterly unpredictable. We even found out that the benchmark suite itself was incorrect with respect to the official number of results. On the opposite end of testing, namely the small but hard test cases, the results are likewise disappointing. Almost all reasoning systems failed at least for one of the cases.

2.1 Benchmarking Scalability

In order to measure the performance of the given systems with respect to large volumes of instance data we have chosen the University Ontology Benchmark (UOBM) [4]. The UOBM extends the well-known Lehigh University Benchmark (LUBM) [5] by adding extra TBox axioms making use of all of OWL Lite (UOBM Lite) and OWL DL (UOBM DL). The ABox is enriched by interrelations between individuals of formerly separated units, which makes it less artificial and more realistic. Please note that our intention is not to simply run this benchmark again. Instead, our main interest is to analyze different settings and outcomes.

The set of reasoning engines within our survey consists of FaCT++ (v1.2.3)[3] [6], KAON2 (2008-06-29)[4], Pellet (v2.0.0RC5)[5] [7] and RacerPro (1.9.3b)[6] [8]. We have used the latest available versions of these systems except for RacerPro, for which we had access to a soon to be released beta version. Hermit (v0.9.3)[7] [9] was initially also within the set of potential candidates but repeatedly produced exceptions while processing the UOBM. A feature matrix of selected system characteristics can be found in Table 2. For all tests in this section we used 64bit versions of the systems, which were running on a 16GB quad-core Linux machine.

Since FaCT++ does not come with a SPARQL or any other conjunctive query language, the set of test systems with respect to query answering only consists of Pellet, RacerPro and KAON2. However, KAON2 is known for its performance decrease in the presence of cardinality restrictions. In fact, the KAON2 engine was two and more magnitudes slower than the other systems. In addition more than 10 GB of main memory were allocated even for the smallest UOBM data set (lite-1). A comparison of KAON2 with the SHER[8] system for the UOBM without cardinality restrictions can be found in [10].

We tested the systems with their standard settings for optimized but complete query answering (e.g. nRQL mode 3) and a warm-up phase of three queries disjoint from the set of benchmark queries. In our comparison, we just computed the number of answers without retrieving them to get measures which are independent of the utilized interface. The systems do not vary dramatically in loading the data. The smallest data set (lite-1) was loaded into KAON2 in 12 seconds,

[3] `http://owl.man.ac.uk/factplusplus`
[4] `http://kaon2.semanticweb.org`
[5] `http://clarkparsia.com/pellet`
[6] `http://www.racer-systems.com`
[7] `http://www.hermit-reasoner.com`
[8] `http://www.alphaworks.ibm.com/tech/sher`

in Pellet in 20 seconds, whereas RacerPro required about 30 seconds. The setup and query preparation time for RacerPro was about 85 seconds compared to Pellet with 9 seconds. Computing the query results for lite-1 was accomplished in less than a second for both Pellet and RacerPro, except for two queries. Pellet needed more than 100 resp. 500 seconds for the queries number 8 and 13 whereas RacerPro required more than 4 resp. 8 seconds for the queries number 2 and 9. Altogether RacerPro completed the whole lite-1 test in roughly 100, Pellet in 680 seconds.

Overall memory consumption was lowest for RacerPro with not more than 3GB for lite-1. Pellet requires up to 4GB of main memory for the same test set. When classifying the TBox before querying, the timings do not vary significantly. The only exception is query number 13 with Pellet only requiring 270 (instead of 550) seconds.

In case of realizing the ABox (as well as classifying the TBox) before querying, it turned out that RacerPro performs slightly better for almost all queries but required more than 16 instead of 8 seconds for query 9. Likewise, Pellet performed dramatically better for three queries (query 2, 8, and 13) with a slight performance decrease for query 4. Realizing the ABox took about 10 minutes for RacerPro and more than one hour for Pellet. We did not succeed in realizing the smallest UOBM lite data set with FaCT++. The system aborted after several minutes, with having allocated all of the 16GB main memory. Interestingly, we were able to realize the small and mid-size UOBM dl data sets (realizing the mid-size data set consumed less than 2.4GB main memory). It seems that the reasoning procedure of FaCT++ for less expressive languages is not as optimized as for the expressive ones. In conclusion, realizing the ABox helps to increase performance for some queries. However, the influence on query answering is somehow unpredictable and due to the computational overhead it is not feasible for larger data sets and was only performed here for comparison.

Interestingly, the number of results for two out of 13 UOBM queries for lite-1 with RacerPro (nRQL mode 3) were different compared to the number of results returned by Pellet as well as the official numbers. For query 10, RacerPro came up with fewer, for query 9 with more answers. When using nRQL's incomplete mode 1 or 2 even eight answers were different, but the performance increased (all queries were processed in less than a second except for query 9).

In a different case, namely query 11 of the dl-5 test set, the official number of query results is obviously wrong. Pellet as well as RacerPro provide 6230 answers here, whereas the UOBM only states 6225 correct results. However, when analyzing the five additional answers it can be easily seen that they meet the query condition. More precisely, all results are correct because of their equivalence to other individuals in the result set. This is due to an individual merge caused by the functional object property "isTaughtBy". Pellet and RacerPro also disagree with the official result on query 15 (also dl-5) with 72 individuals. Both reasoner, however, return no answer at all.

Based on our findings we conclude that it is not only difficult to write expressive and scalable reasoning engines but also to generate sophisticated benchmark

Table 1. Correctness Spot Tests

no.	expressivity	C	P	D	I	FaCT	Racer	Pellet	HermiT	KAON2
1b	$\mathcal{SHIN}$	6	7	0	0	+	time	time	+	time
2a	$\mathcal{SHN}$	8	9	0	0	+	+	+	time	time
2b	$\mathcal{SHN}$	8	9	0	0	+	+	time	time	time
9	$\mathcal{SHF}$	7	7	0	0	-	+	+	+	+
28	$\mathcal{ALC}$	33	7	0	0	+	+	+	time	+
29b	$\mathcal{SHIF}$	12	6	0	0	-	+	+	+	+
30	$\mathcal{SHOIF}$	23	6	0	41	+	(+)	error[9]	+	n.a.
35	$\mathcal{ALCOI}$	17	7	0	9	memory	(+)	+	+	n.a.

suites. More than that, conducting benchmarks requires a careful interpretation of the results from different view points and for different requirements.

2.2 Sound- and Completeness Spot Tests

Reasoning with OWL Lite as well as OWL DL is known to be of high worst-case complexity. However, trading soundness/completeness for efficiency is not a solution, in our opinion. Instead, application designers should carefully choose their language fragment and size of ontology in order to be able to pick the right approach resp. system. Implementors of reasoning engines should, on the other hand, take special care about creating a reliable system in terms of soundness and completeness (which is, for sure, a formidable challenge). Otherwise, performance results of different systems are not comparable at all. Even if a bug-free reasoning system may never be available, our experiences in intensively using various engines have revealed a number of disappointing results. Some systems fail to answer even small or trivial test cases, some failures tend to re-appear in subsequent system versions, or termination varies not only from system to system but also from release to release.

As a consequence, we tested our systems with an empirical evaluation using spot tests which are intentionally designed to be hard to solve but small in size. They try to meter the correctness of the reasoning engines with respect to inference problems of selected language features and were published two years ago [2]. The tests were conducted with the systems listed in Table 2. We used FaCT, Pellet, Hermit via the OWL-API, whereas KAON2 and RacerPro were tested via their native interfaces for loading OWL files. Surprisingly, almost all reasoners failed in some test cases by providing either wrong results or were not able to find a solution within 10 minutes. Table 1 shows a selection of these test cases, all ran on a Linux maschine (ubuntu 2.6.17, 16GB ram, Java 1.6, gcc 4.0.3):

[9] Varies with respect to platform (Linux resp. MacOS) and Java version (1.5 resp 1.6) from time-out to an error caused by an exception.

Table 2. Reasoner Feature Matrix

	OWL-API	Jena	DIG	OWLlink	native API	language support	retraction	incr. reas.	SWRL supp.	query language	query entailm.	available license	impl. lang.
FaCT++ 1.2.3	X		X		KRSS like	OWL 2 $\mathcal{SROIQ}$(D)						LGPL v2.1	C++
Pellet 2.0.0RC5	X	X	X			OWL 2 $\mathcal{SROIQ}$(D)	X	X	X	SPARQL		AGPL v3 & Commercial	Java
RacerPro 1.9.3b	X		X	X	KRSS like	OWL 1 $\mathcal{SHIQ}(\mathcal{D})$	X		X	nRQL	X	Academic & Commercial	Lisp
KAON2 2008-6-29			X		KAON	OWL 1 $\mathcal{SHIQ}(\mathcal{D})$			X	SPARQL		Academic & Commercial	Java
HermiT 0.9.3	X				KAON	OWL 2 $\mathcal{SHOIQ}(\mathcal{D})$						GPL	Java

In the first column, each test case has been assigned an identifier.[10] The subsequent columns list the expressivity, the number of classes (C), object properties (P), datatype properties (P), and individuals (I) of the respective test case. A "+" for a specific system means that the system could provide the correct answer (typically satisfiability/unsatisfiability of a class), whereas round brackets denote that the result was computed on the basis of approximate reasoning. Furthermore, "time" indicates that the system could not provide an answer within ten minutes. Since our test environment provided more than 16GB of main memory an "out of memory" error did only occur once within our time frame of 10 minutes. A "-" signals that the system returned the wrong result. To identify potential modeling patterns that are supposed to slow down reasoning, we fed all the test cases into Pellint[11] v0.2 [11]. Hereby, no performance barriers have been reported.

It is worth noting that the results of our spot test often change from version to version for FaCT++ and Pellet and even from platform to platform resp. environment (operating system and/or Java version). For instance, previous versions were able to solve 1b (Pellet) or 2b (FaCT++) but did not terminate or failed with others. In addition, FaCT++ v. 1.2.1 on Mac OS X (gcc 4.0.1) succeeds for test case 2b but fails for 1b. With version 1.2.0 it is the other way round. The newest Pellet release candidate (2.0.0RC5) is able to solve test case 28 within seconds whereas previous versions did not. Interestingly, various versions of Pellet randomly abort with an exception on test case 30 (whereas Pellet 1.4 produces a time-out). Test case 30 is the only one using all of OWL DL and

[10] The corresponding OWL file can be found at `http://www.informatik.uni-ulm.de/ki/Liebig/reasoner-eval/no.owl`

[11] `http://pellet.owldl.com/pellint`

cannot be handled by KAON2 and only in an approximative way by RacerPro. Again, the MacOS X version of FaCT++ does not terminate on test case 30.

Please note that all test cases above are within OWL 1 and even only a fraction of OWL Lite in some cases. The upcoming OWL 2 adds further language constructs which makes reasoning even more difficult. Extrapolating the given results indicates that reliable OWL 2 reasoning engines might still need some more time to mature.

3 Reasoning in Practice

IYOUIT[12] [12], a mobile community service in the field of context-awareness makes use of formal representations to reason about gathered data. IYOUIT allows people equipped with an ordinary mobile handset like the gPhone to instantly share personal experiences with others while on the go. The mobile application is connected to a network of components on the Internet. All data gathered by the mobile handset, such as the current location or nearby Bluetooth beacons, are sent to the appropriate component, which stores and further processes this information. One of the main tasks of each component is to abstract from low-level sensor data (e.g., location traces) to corresponding qualitative representations (e.g., important places of a user). Once theses abstractions are linked to concepts formalized in OWL ontologies, reasoning is applied to derive additional information. In the following, two of these reasoning use cases are highlighted, both with different requirements on the expressiveness of the underlying ontology and temporal constraints on the overall reasoning process.

All relationships in the IYOUIT user community are represented within a social ontology to allow for expressing social relationships between users. Ontology based reasoning is applied to maintain the consistency of the social network and to make implicit relations explicitly available. The structure of the complemented social network is in turn used for privacy control.

We performed several tests to find the appropriate combination of reasoning engine and ontology modeling style that complies with the requirements on correctness as well as performance. To ensure comparable results, we connected all reasoning engines via the standard Java-based OWL API[13] [13] using the corresponding connector.[14] In addition, we analyzed the performance of the HTTP-based DIG protocol via the OWL API DIG connector, OWLlink[15] [14] and RacerPro's native TCP interface. All tests have been performed with an Apple MacBook Pro 2,33 GHz Intel Core 2 Duo notebook computer with 3GB of memory running Mac OS X 10.5.5 with Java 5.0 32bit and Java 6.0 64bit,

[12] `http://www.iyouit.eu`

[13] We used the SVN version of the OWL API from `http://owlapi.sourceforge.net` as of 9.12.2008 for all experiments.

[14] Reasoners analyzed in Section 2 that do not implement all required functions of the OWL API interface yet, were not considered in the following tests.

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

respectively. The reasoning engines were initialized with their default configuration and the Java Virtual Machine has been configured as follows: The maximum stack size for each thread has been set to 4MB, the minimum heap space to 30MB and the maximum heap space has been limited to 200MB. Before the actual measurements have been recorded, five consecutive runs were applied to warmup the Java hotspot compiler and to initialize the reasoner. In total, 100 runs were performed to average out minimum and maximum measurements and to simulate the actual real-world setting as part of a server infrastructure.

3.1 Social Ontology

To model qualitative social relationships we developed a social ontology for the IYOUIT system. Formalized in OWL-Lite, a fragment of OWL 1, it fulfills the requirements on expressivity (in modeling the social network) and decidability. Complying with the Description Logic $\mathcal{SHIF}$, the social ontology makes use of an object-property hierarchy with transitive, inverse and functional properties (cf. Figure 1). Social relationships are represented as instantiations of those object-properties, whereas entities that represent IYOUIT users are formulated as instances of the class `Person`. In total, the social ontology contains, 6 primitive concepts, 1 disjoint class axiom, 20 object properties and 110 individuals with 280 object property assertion axioms.

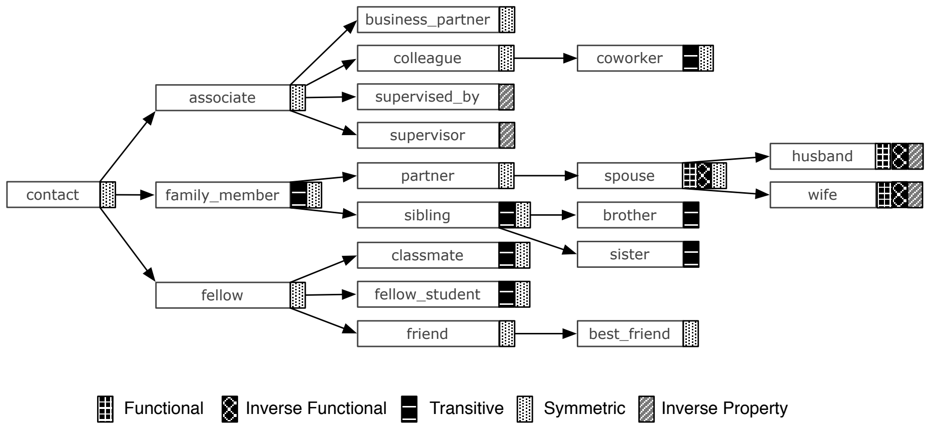

Fig. 1. Social Object Properties

We initially defined a set of test cases to verify that our social ontology captures the intended semantics.[16] The first two queries, encoded in Test A and Test B, analyze the inferred object property structure by requesting the direct

[16] The ontologies for all tests in this section can be found at `http://www.informatik.uni-ulm.de/ki/Liebig/reasoner-eval/IYOUIT.zip`

and the full set of sub-properties of the property `contact`, respectively. Please note that for Test A and B an earlier and slightly larger version of the social ontology with in total 74 object properties has been used. The last query, Test C, requests the total set of object property relationships of one specific individual. Here, the optimized social ontology with its 20 object properties has been used (cf. Figure 1). This ontology and the respective query can also be found in the actual real-world application within IYOUIT.

Interestingly, different reasoning engines returned different results for our queries. Table 3 lists the number of results returned by different engines for all 3 test cases. To the best of our knowledge, only the results returned by RacerPro v1.9.3b are correct, whereas all other results are wrong, which has also been confirmed by the respective developers. Earlier versions of RacerPro returned wrong results as well, similar to FaCT++ for versions in between 1.1.9–1.1.11 that returned the same wrong results as the latest FaCT++ version 1.2.1. Version 1.2.0 is the only version of FaCT++ that provided the correct results for case A and B (case C could not be tested via the OWL API since the required interface function is only supported in v1.2.1). Pellet version 1.5.0 and 1.5.1 suffer from a typo in the OWLAPI interface implementation and return only the direct sub-properties in Test case B. For Pellet 2.0.0RC1, the result set for Test A and B actually depends on the JavaVM version. Here, the number of results indicated before the slash are obtained with Java v1.5, those after the slash with v1.6.

In summary, we were astound by the number of issues (sometimes reoccurring in later versions after having been fixed before) produced by our relatively simple ontology. It seems that the use of inverse properties accounts for the complications in the reasoning kernel in most cases, while other problems were just caused by implementation errors in the corresponding interface functions.

Table 3. Object Property Inferences

	Test A	Test B	Test C
FaCT++ 1.2.1	12	50	53
Pellet 1.4	10	10	38
Pellet 2.0.0RC1	10/9	44/46	71
Pellet 1.5.2 & 2.0.0RC3	10	50	71
RacerPro 1.9.3b	10	50	72

3.2 Object Property Reasoning

Social reasoning mechanisms are applied to complement the social network of IYOUIT users and to maintain the consistency of explicitly given social relations. To find the most appropriate reasoner for the social network reasoning, four different reasoning engines, namely RacerPro 1.9.3 beta, Pellet 1.5.2, the latest Pellet release candidate version 2.0.0RC3 and FaCT++ 1.2.1, were compared via the OWL API. To measure the practical reasoning performance and to trigger

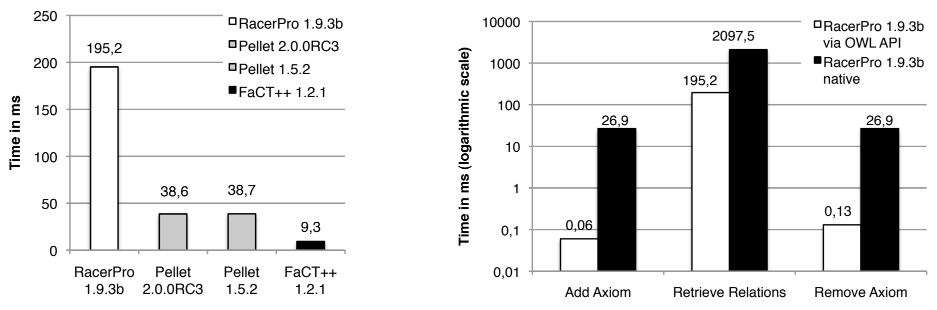

Fig. 2. Object Property Reasoning Performance

the actual classification process, an additional relationship property axiom has been added to the already defined 280 object property assertion axioms of the social ontology. Next, all relationships were retrieved from the reasoner and finally, the initially added relationship has been removed again. While adding and removing axioms has been performed very fast by all reasoning engines, significant differences were observed in retrieving the actual inference result. On the left-hand side of Figure 2, the results of this retrieval process are shown in more detail. With an average of 9,3 milliseconds, FaCT++ has been the fastest engine, followed by Pellet with 38,6 milliseconds. Here, no significant performance difference was found when comparing Pellet 1.5.2 with the latest version 2.0.0RC3. With 195,2 milliseconds, RacerPro 1.9.3b took considerably longer to answer the given query. Performance aside, RacerPro has been the only system that returned the correct result (1620 relationships). Pellet was missing exactly one relationship and FaCT++ only returned 1440 relationships (cf. Section 3.1).[17] As a result, a concluding performance assessment cannot be made as long as not all reasoning engines do return the correct result.

In a second test, we compared two currently available interfaces for RacerPro, the OWL API and its native JRacer interface, both connecting the reasoner and the Java application via TCP. Again, adding and removing axioms took considerably less time than retrieving the inference result. To our surprise, the OWL API connection performed dramatically better than the native connection for both, the inference retrieval as well as the axiom retraction. As summarized on the right-hand side of Figure 2, transferring all relationships from RacerPro was finished within 195,2 milliseconds in case of the OWL API and 2097,5 milliseconds in case of the native JRacer connection. One feasible explanation, which has also been confirmed by the implementors of RacerPro, could be the fact that the RacerPro OWL API connector is based on an optimized TCP protocol, while the JRacer connection still relies on an older and thus un-optimized version of the latter.

[17] In its latest release candidates RC4 and RC5, Pellet now also returns the correct result. Two newer versions of Fact++, however, do not include a respective fix.

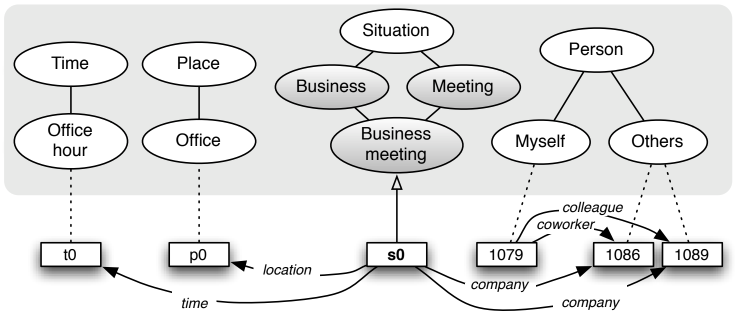

Fig. 3. Situation Description

3.3 Dynamic Classification

The situation estimation component [15] of the IYOUIT system constantly retrieves abstractions of the latest sensor data produced by the IYOUIT mobile client such as the actual place of a user, a qualitative description of the current time and the people in proximity. Its task is to compute an abstract description of a user's situation based on live context data and static profile information, including all established qualitative social relationships.

The set of OWL DL ontologies used to realize this situational reasoning has been refined step by step. The initial version, in the following referred to as Ontology C, is within $\mathcal{SHOIN}$ and consists of 270 (cyclic) classes, 7 CGIs, 135 object properties, 65 datatype properties and 333 individuals. It is distributed over 9 component ontologies including the social ontology (without its individuals and object property assertion axioms) discussed above and the standard time-entry ontology [16]. The second version of the ontology, Ontology B, was derived from Ontology C by removing all remaining individuals and nominals, breaking the cycles between classes, reducing the number of domain/range restrictions, datatype properties and inverse functional declarations, adding additional classes and disjoint class axioms, replacing the time-entry ontology by a simpler time ontology and reducing the number of components. This results in 324 classes, 0 CGIs, 116 object properties, 43 datatype properties and 0 individuals distributed over 4 component ontologies, reducing the complexity to $\mathcal{SHIN}$. The latest ontology, Ontology A, was derived from Ontology B by reducing the number of object properties, classes and disjointness axioms, by removing all datatype properties, most of the remaining domain/range restrictions and all annotations, resulting in 137 classes, 0 CGIs, 24 object properties, 0 datatype properties and 0 individuals, distributed over 4 component ontologies in $\mathcal{SHIN}$.

Once the ontologies are loaded into the reasoner and all qualitative data used to characterize a users situation is gathered from the IYOUIT network, the data is translated into an OWL situation description, represented as individuals of the corresponding classes within the ontology (cf. Figure 3). A situation individual `s` of class `Situation` is associated with individuals representing the abstract time and location of the user, as well as the buddies detected in proximity using object property axioms. Additionally, object property axioms are defined between the considered user (a member of the `Myself` class) and the detected nearby buddies corresponding to the established social relations as discussed above. Based on the axiomatization of the subclasses of `Situation`, the derived situation of `s` can be retrieved by asking the reasoning engine for its direct types. In a final clean-up step, the situation individual `s` is again removed from the ontology, together with all associated individuals and object property axioms.

In case of the situational reasoning, our main interest is to compare the effect of the utilized interface between the reasoner and the application, since the actual inference problem is rather simple. Comparing the performance of one classification task formulated with ontologies of varying size and complexity, we examine the influence of the ontology modeling and the effect of the potential use of (large) standard ontologies. Furthermore, we analyze the effect of retraction and incremental reasoning by defining one distinct classification task in the following three steps: a) establishing a situation description by adding the corresponding axioms; b) requesting the classification result; c) removing the axioms describing the situation again.

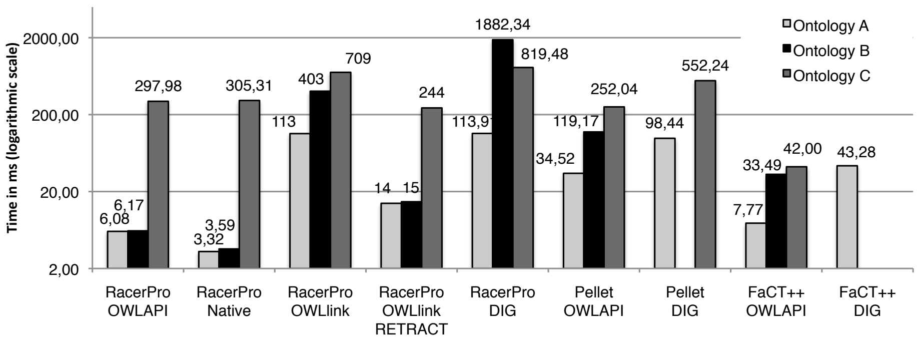

Fig. 4. Interface Performance

Figure 4 summaries the results we obtained with the latest available versions of RacerPro (v1.9.3b), FaCT++ (v1.2.1) and Pellet (v2.0.0RC3). The reasoners were connected via the OWL API, via DIG over HTTP using the corresponding DIG OWL API connector, and, for RacerPro, additionally via the native JRacer connection over TCP and the OWLlink interface over HTTP. While Pellet and FaCT++ run within the same memory partition with the OWL API interface being directly connected via the corresponding connector (in case of FaCT++ using its JNI interface), the RacerPro OWL API connector accesses RacerPro via its native TCP interface.

Not surprisingly, the actual performance largely depends on the complexity of the underlying ontology. However, the effect of the selected ontology seems to influence the performance of RacerPro differently than the other reasoners.

RacerPro, connected via TCP and with support for retraction, outperformed the in-memory connected reasoners Pellet and FaCT++ for the less complex and non-cyclic ontologies. The native RacerPro connection, without the additional overhead caused by the OWL API, showed an even better performance. However, in earlier experiments we discovered that predecessors of Racer v1.9.1 performed slower from run to run, an issue that seems to have been solved in later versions. The DIG interface fails to properly transmit the ontology to the reasoner as some constructs (like cardinality restrictions on datatype properties) are not supported by this protocol.[18] This results in cyclic knowledge bases for all three variants of the ontologies, a bad overall performance, in OutOfMemory errors for Pellet in some cases and DIGReasonerExceptions for FaCT++. OWLlink, the successor of DIG, allows to transmit all of OWL 2 and supports retraction. OWLlink via HTTP using retraction outperforms even Pellet's in-memory connection. On the positive side, different versions of Pellet and FaCT++ connected via the OWL API (cf. Figure 5) show a steady improvement of performance during the last 1.5 years, especially for the more complex versions of the ontology.

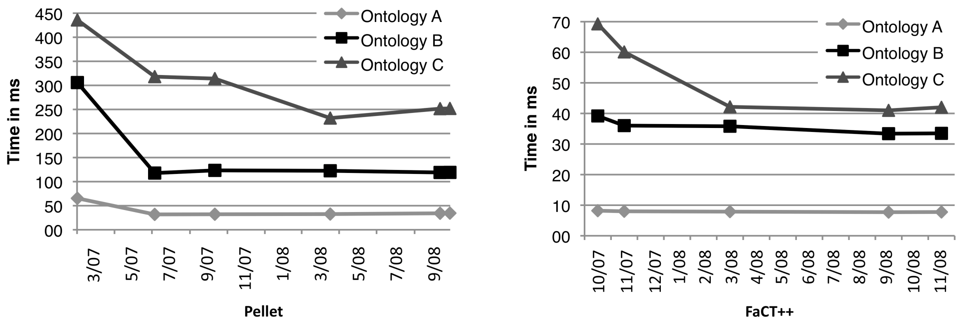

Fig. 5. Reasoner Development

4 Conclusions

By applying Semantic Technologies in real-world applications, complex facts can be described by a well-defined model instead of writing application specific code. A major advantage of this approach is the clear separation of the application logic from the underlying model, which (in principle) reduces the risk of system failures. However, choosing the appropriate combination of a reasoning engine, a communication interface and expressivity of the utilized ontology is an underestimated complex and time-consuming task. To evaluate the performance of the selected components, appropriate test cases seem to be inevitable. However,

[18] Furthermore, the latest OWL API version translated exact cardinality restrictions into invalid DIG syntax. An issue we fixed before running the experiments.

what is known for being a good solution in one case, might as well be an inappropriate solution in another case. Standard test environments such as the one introduced by Gardiner et al. [17] point in the right direction as they allow for specifying and verifying the expected results. However, this approach is based on DIG, and therefore fails to transfer the entire set of OWL axioms to the reasoning engine. Isolating condensed test cases when either dealing with huge knowledge bases or in case the correct result is not known beforehand is a challenging issue in practice. Here, the recent advancement of the OWL API is most welcome, since it makes comparing different reasoning engines and ontologies far easier without having to adapt the core application logic. Likewise, OWLlink, the recently introduced successor of DIG, offers an implementation-neutral and extensible protocol for the communication with diverse OWL reasoning systems (also supporting retraction). Yet, deploying applications as part of a Web-service based server infrastructure, native reasoner connections via TCP can still have an advantage compared to in-memory solutions. Here, the actual application is not affected in case the reasoning engine might not be responsive any more.

Another noticeable trend in recent years is concerned with the improvement of implemented optimization strategies of most reasoning engines. However, we were wondering why the addition of incremental consistency checking under syntactic Abox updates in Pellet v1.5.0 [18] and the newly introduced incremental classifier of Pellet 2.0.0RC1 [19] did no have a larger impact on the performance of our incremental situation classification task. At least for some cases, these optimizations result in more efficient reasoning tasks. Yet, we found that for relatively simple reasoning use cases in which the actual classification takes only a minimal amount of time, the inference retrieval is by far the most expensive task in the whole process. Part of the problem lies in the fact that there is no appropriate interface that would allow for retrieving only inferred axioms from the reasoning engine. As a result, the application itself needs to retrieve the entire inference result, including all told axioms, to compute only newly inferred axioms. To maintain correct inference results and to ensure efficient reasoning tasks, only lightweight, specifically designed ontologies were considered. Utilizing standard ontologies such as DOLCE [20] or OpenCyc[19] to profit from a well-defined cross domain vocabulary concurrently increases complexity with respect to the inference traceability and might as well have a negative impact on the overall reasoning performance since reasoning in OWL is not modular.

References

1. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P., Sattler, U.: OWL 2: The next step for OWL. Web Semantics: Science, Services and Agents on the World Wide Web **6** (2008) 309–322
2. Liebig, T.: Reasoning with OWL: System Support and Insights. Technical Report TR-2006-04, Ulm University, Germany (2006)

[19] `http://www.opencyc.org`

3. 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'07). (2007) 296–310
4. Ma, L., Yang, Y., Qiu, Z., Xie, G., Pan, Y., Liu, S.: Towards a Complete OWL Ontology Benchmark. In: Proc. of the 3rd European Semantic Web Conference (ESWC'06). Volume 4011 of LNCS., Springer (2006) 125–139
5. Guo, Y., Pan, Z., Heflin, J.: An Evaluation of Knowledge Base Systems for Large OWL Datasets. In: Proc. of the 3rd Int. Semantic Web Conference (ISWC'04), Hiroshima, Japan (2004) 274–288
6. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 636–647
7. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics **5** (2007) 51–53
8. Haarslev, V., Möller, R.: Description of the RACER system and its applications. In: Proc. of the Int. Workshop on Description Logics. (2001)
9. Shearer, R., Motik, B., Horrocks, I.: HermiT: A Highly-Efficient OWL Reasoner. In: Proc. of the OWL Experiences and Directions Workshop at the ISWC'08. (2008)
10. Dolby, J., Fokoue, A., Kalyanpur, A., Ma, L., Schonberg, E., Srinivas, K., Sun, X.: Scalable conjunctive query evaluation over large and expressive knowledge bases. Technical report, IBM Watson Research Center (2008)
11. Lin, H., Sirin, E.: Pellint – A Performance Lint Tool for Pellet. In: Proc. of the OWL Experiences and Directions Workshop at the ISWC'08. (2008)
12. Böhm, S., Koolwaaij, J., Luther, M., Souville, B., Wagner, M., Wibbels, M.: Introducing IYOUIT. In: Int. Semantic Web Conf. (ISWC'08), October 27–29, 2008. Volume 5318 of LNCS., Springer (2008) 804–817
13. Horridge, M., Bechhofer, S., Noppens, O.: The OWL API. In: Proc. of the 3rd OWL Experiences and Directions Workshop at the ESWC'07. (2007)
14. Liebig, T., Luther, M., Noppens, O., Rodriguez, M., Calvanese, D., Wessel, M., Horridge, M., Bechhofer, S., Tsarkov, D., Sirin, E.: OWLlink: DIG for OWL 2. In: Proc. of the OWL Experiences and Directions Workshop at the ISWC'08. (2008)
15. Luther, M., Fukazawa, Y., Wagner, M., Kurakake, S.: Situational reasoning for task-oriented mobile service recommendation. The Knowledge Engineering Review **23** (2008) 7–19
16. Pan, F., Hobbs, J.: Time in OWL-S. In: Proceedings of the AAAI Spring Symposium on Semantic Web Services, California, Stanford University (2004) 29–36
17. Gardiner, T., Horrocks, I., Tsarkov, D.: Automated Benchmarking of Description Logic Reasoners. In Parsia, B., Sattler, U., Toman, D., eds.: Proc. of the Int. Workshop on Description Logics (DL'06). Volume 189., CEUR.org (2006) 167–174
18. Halaschek-Wiener, C., Parsia, B., Sirin, E.: Description logic reasoning with syntactic updates. In Meersman, R., Tari, Z., eds.: OTM Conferences. Volume 4275 of LNCS., Springer (2006) 722–737
19. Grau, B.C., Halaschek-Wiener, C., Kazakov, Y.: History matters: Incremental ontology reasoning using modules. In: Proc. of the 6th Int. Semantic Web Conf. and 2nd Asian Semantic Web Conf. (ISWC/ASWC'07). Volume 4825 of LNCS., Springer (2007) 183–196
20. Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., Schneider, L.: Sweetening ontologies with DOLCE. In: Proc. of the 13th Int. Conf. on Knowledge Engineering and Knowledge Management. Volume 2473 of LNCS., Springer (2002) 166–181