

# FaCT and OilEd Clients

Sean Bechhofer  
University of Manchester  
Kilburn Building  
Oxford Road  
Manchester M13 9PL  
email: seanb@cs.man.ac.uk

with contributions from Daniel Oberle<sup>1</sup>, Raphael Volz<sup>1,2</sup>, Ralf Möller<sup>3</sup> and Peter Crowther<sup>4</sup>

<sup>1</sup> University of Karlsruhe, <sup>2</sup> FZI, Karlsruhe, <sup>3</sup> Fachhochschule Wedel, <sup>4</sup> Melandra Limited



<b>Identifier</b>	Del 10
<b>Class</b>	Deliverable
<b>Version</b>	1.0
<b>Date</b>	August 28, 2003
<b>Status</b>	Draft
<b>Distribution</b>	Public
<b>Lead Partner</b>	VUM

## **WonderWeb Project**

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks  
The Victoria University of Manchester  
Department of Computer Science  
Kilburn Building  
Oxford Road  
Manchester M13 9PL  
Tel: +44 161 275 6154  
Fax: +44 161 275 6236  
Email: [wonderweb-info@lists.man.ac.uk](mailto:wonderweb-info@lists.man.ac.uk)

# Contents

<b>Executive Summary</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Integrating FaCT . . . . .	2
1.2 Integrating OilEd . . . . .	2
<b>2 The FaCT System</b>	<b>2</b>
<b>3 The DIG Interface</b>	<b>3</b>
3.1 Overview . . . . .	4
<b>4 Implementing the DIG protocol for FaCT</b>	<b>5</b>
4.1 Architecture . . . . .	5
<b>5 The OilEd ontology editor</b>	<b>6</b>
5.1 Reasoning with OilEd . . . . .	7
5.2 Reasoner Integration (pre-KAON) . . . . .	8
<b>6 Integrating DIG and KAON</b>	<b>8</b>
<b>7 Integrating OilEd and DIG via KAON</b>	<b>8</b>
<b>8 Integrating OilEd and an RDF store via KAON</b>	<b>9</b>
<b>9 Conclusions</b>	<b>11</b>

## **Executive Summary**

This document describes the integration of the FaCT reasoner and the OilEd ontology editor with the KAON server. FaCT is a Description Logic reasoner that supports in particular the classification of, and identification of inconsistencies within, ontologies. OilEd is an editor that allows the user to construct and manipulate DAML+OIL and OWL ontologies and which uses a reasoner to classify and check consistency of ontologies.

# 1 Introduction

In order to demonstrate the component-based architecture of the KAON Server, the objectives of WonderWeb include the integration of a number of existing tools and services. This deliverable discusses the integration of two such tools – the FaCT Description Logic reasoner [13] and the OilEd ontology editor [3].

These two tools illustrate different integration scenarios. FaCT is a tool that effectively offers a service that may be of interest to other client tools, and is thus best integrated as a component, while OilEd can be considered purely as a client of the server, making use of the services supplied by components such as FaCT or the Triple Client described in [18].

## 1.1 Integrating FaCT

Integrating the FaCT reasoner involves two steps:

1. Defining an interface to the reasoner, e.g. specifying the operations that the reasoner provides.
2. Providing this interface as a component within the KAON Server.

The first task has been achieved through the specification of an interface known as DIG. The DIG specification is a proposal for a standard Description Logic (DL) reasoner interface protocol that has been developed in collaboration with a number of other DL system implementors. This is described in Sections 2 and 3 and 4 and allows us to integrate not only the FaCT reasoner, but also other DL reasoners supporting the DIG protocol. The second task requires the wrapping of the component providing the DIG service as a component within the KAON framework. This is described in Section 6.

## 1.2 Integrating OilEd

Integrating OilEd is a somewhat simpler matter, and effectively involves rewriting code that connects to components such as a reasoner and RDF store to connect and interact with the components via the KAON server. This is described in Sections 7 and 8.

# 2 The FaCT System

The FaCT System [13] is a result of several years of research into the optimisation of tableaux subsumption algorithms. FaCT (Fast Classification of Terminologies) is a Description Logic (DL) classifier that can also be used for modal logic satisfiability testing. Research on extensions to, and improvements on, the implementation of FaCT is an ongoing activity – a forthcoming WonderWeb project deliverable [9] will describe recent developments.

Fact's most interesting features are:

- its expressive logic ;

- its support for reasoning with arbitrary knowledge bases (i.e., those containing general concept inclusion axioms);
- its optimised tableaux implementation (which has now become the standard for DL systems).

In addition, interfaces to FaCT have been developed (CORBA-FaCT [5] and the DIG interface [7]) that facilitate the use of the reasoner within applications. The DIG interface is discussed in more detail below.

FaCT itself is effectively a reasoner that performs satisfiability testing. An ontology or knowledge base is given to the reasoner as a collection of axioms, and queries concerning the satisfiability of concept expressions can then be presented to the reasoner. How we actually make use of this in an application such as OilEd is discussed below.

### 3 The DIG Interface

Most description logic (DL) systems present the application programmer with a functional interface, often defined using a Lisp-like syntax. Such interfaces may be more or less complex, depending on the sophistication of the implemented system, and may be more or less compliant with a specification such as KRSS [17].

The Lisp style of the KRSS syntax reflects the fact that Lisp is still the most common implementation language for DLs. This can create considerable barriers to the use of DL systems by application developers, who often prefer other languages (in particular the currently ubiquitous Java), and who are becoming more accustomed to component based software development environments. This is of increasing importance given current interest in Web Services and service based architectures.

In such an environment, a DL might naturally be viewed as a self contained component, with implementation details, and even the precise location in which its code is being executed, being hidden from the application [1]. This approach has several advantages: the issue of implementation language is finessed; the API can be defined in some standard formalism intended for the purpose; a mechanism is provided for applications to communicate with the DL system, either locally or remotely; and alternative DL components can be substituted without affecting the application.

This approach was adopted in the CORBA-FaCT system [6], where a CORBA interface was defined for a description logic reasoner. This wrapping of the FaCT reasoner facilitated the successful use of the reasoner in applications such as OilEd [4] and ICOM [10].

Although useful, CORBA-FaCT suffered from a number of inadequacies and idiosyncracies. The concept language does not cover concrete domains, and as the interface was largely based on the FaCT reasoner (which did not support an A-box at the time), functionality relating to Aboxes was missing. In addition the identifiers allowed were closely tied to the underlying Lisp implementation (case insensitive strings of essentially alphanumeric characters). This introduces problems when trying to reason over languages such

as DAML+OIL where classes are referred to using URIs<sup>1</sup>.

The RACER [12] system adopted a slightly different mechanism, providing a socket based interface for use by client applications. This again provides a language neutral API, but the lower-level interface places more onus on the client programmer.

The DL Implementation Group (DIG)<sup>2</sup> is a self-selecting assembly of researchers and developers associated with implementations of Description Logic systems. As part of the activity of DIG, a new interface for DL systems has being defined.

The specification described here should be considered as a *Level 0* specification – in its current version it contains just enough functionality to enable tools such as OilEd [4] to use a reasoner. Later version of the specification may address issues such as stateful connections, transactions, reasoner preferences and so on. There may also be the possibility of extending the concept language.

There is nothing inherently new in the specification – it is effectively an XML Schema for a DL concept language along with ask/tell functionality. Along with the definition of this interface, however, there is a commitment from the implementors of the leading DL reasoners (FaCT [14], RACER [12] and Cerebra<sup>3</sup>) to provide implementations conforming to the specification. This will truly allow us to build plug and play applications where alternative reasoners can be seamlessly integrated into our systems.

### 3.1 Overview

The specification essentially consists of an XML Schema [20] describing the expressions of the concept language, the available tell and ask operations along with the expected responses and administrative information.

These messages are then passed across to a DIG reasoner using HTTP [11] as the underlying protocol. In this respect, we borrow from other initiatives such as SOAP<sup>4</sup> and XML-RPC<sup>5</sup> which have both built messaging protocols using XML on top of HTTP. The use of HTTP allows client (and server) developers to use existing libraries for implementation<sup>6</sup>. We are not, however, strongly wedded to the use of HTTP as the underlying protocol. A richer mechanism may be adopted in the future. Indeed, alternative mechanisms for communication may be employed for efficiency – for example, communication between a client and a reasoner may be possible using some kind of in-memory link.

Full details of the DIG protocol can be found in [7].

---

<sup>1</sup>Of course such problems are not insurmountable, but do provide barriers to the ease of use of the systems.

<sup>2</sup>See <http://dl.kr.org/dig> for more details of DIG.

<sup>3</sup><http://www.networkinference.com/>

<sup>4</sup><http://www.w3.org/TR/SOAP/>

<sup>5</sup><http://www.xmlrpc.com/>

<sup>6</sup>Although this should not be a prime motivation for the use of the protocol, building on top existing work is likely to improve the chances of the DIG Interface being used.

## 4 Implementing the DIG protocol for FaCT

FaCT is written in Lisp, and the core reasoner offers a basic command-line interface. In order to support the DIG protocol as described in Section 3, we need to supply a wrapper or interface layer that translates DIG requests to the appropriate FaCT requests.

Java appears to be the current “language of choice” for the implementation of graphical user interfaces (GUIs) and client tools, so the implementation of this wrapper has been in Java. This does not mean, however, that clients can only communicate with a DIG reasoner using Java – the HTTP based protocol ensures that client communication is effectively language neutral. The use of Java for the wrapping does offer alternative mechanisms for communication when implementing clients in Java – in particular via a direct in-memory connection as described below, which may improve efficiency and reduce communication overhead.

### 4.1 Architecture

To isolate clients from any implementation details, a `dig.Reasoner`<sup>7</sup> interface is defined. This supplies a single method that takes a request and then returns a response. Implementations of this interface can then supply the required functionality through whichever means they wish.

The FaCT Reasoner can then be wrapped with an appropriate `dig.FaCTReasoner` wrapper that converts DIG requests to FaCT reasoner calls. This is shown in Figure 1. The class `dig.DIGReasoner` provides an implementation of the `dig.Reasoner` interface

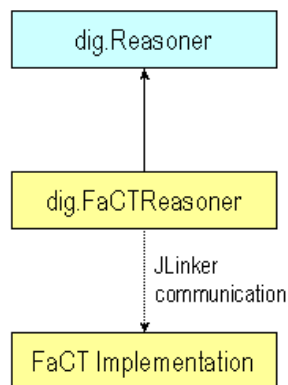


Figure 1: FaCT DIG Architecture

that processes the requests through a translation to the lisp formats understood by the FaCT Lisp implementation. The actual communication with FaCT is achieved through

---

<sup>7</sup>Class and Interface names here are not exactly those used in the implementation – for example the full package names are not given. Full details of implementations will be provided in documentation accompanying the software.

use of the JLinker facility provided by Allegro<sup>8</sup>.

The use of the interface within the client implementation, however, means that alternative implementations can be provided. For example, a `dig.HTTPReasoner` can be defined which simply passes the requests on to a remote reasoner at a particular location using the HTTP protocol as discussed in Section 3. An extended architecture diagram is shown in Figure 2. In this figure we see the `dig.HTTPReasoner` making use of an HTTP connection to some DIG Servlet implementation (which may itself employ a DIG reasoner to perform the reasoning).

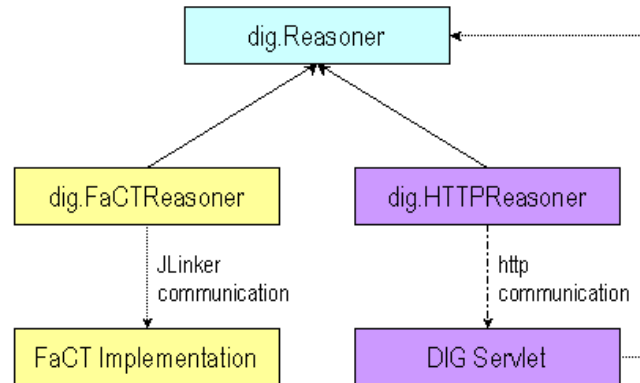


Figure 2: Architecture extended with HTTP Reasoner

Client implementations can then use the `dig.DIGReasoner` interface, and a suitable implementation object can be supplied at run-time. This means that the implementor of a client application such as OilEd can delay the choice of communication mechanism until run-time.

## 5 The OilEd ontology editor

Tools allowing construction and maintenance of ontologies will play a key part in the provision of ontology infrastructure in the Semantic Web.

OilEd [3] is a simple ontology editor that allows the user to create, edit and manipulate OWL ontologies. OilEd is not a fully fledged editing environment, but instead provides a rather lightweight tool for ontology editing. The functionality has been sufficient, however, for many purposes, and the tool has been used extensively in research groups, commercial enterprises and for education.

OilEd is discussed at length elsewhere [3], so we will not go into detail here other than to describe the key aspects of the tool.

A aspect of OilEd which is of interest, is that unlike current versions of existing frame-based ontology editors such as Protégé or OntoEdit, it supports the full expressive power

<sup>8</sup>See <http://www.franz.com/support/documentation/6.2/doc/jlinker.htm> for details of JLinker.

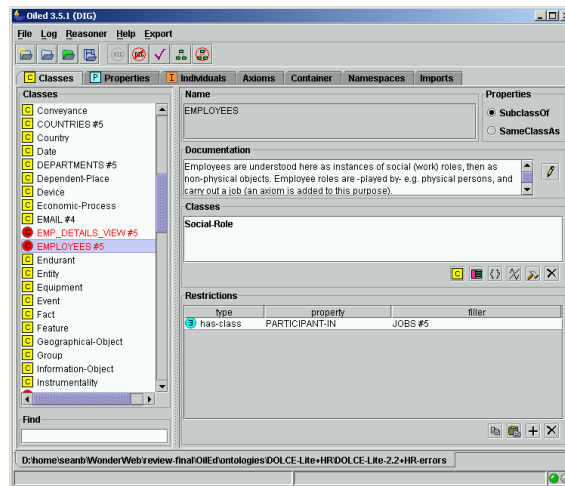


Figure 3: OilEd Screen Shot

of languages like DAML+OIL and OWL. Thus users can produce concept descriptions that make use of, for example, arbitrary boolean combinations of expressions and explicit quantification. However, the user is not forced into using complex OWL constructs, and the editor provides a simple “frame-based” view of the ontology whenever possible. For example, Figure 3 shows a concept `EMPLOYEES` as being defined as a subclass of `Social-Role` that is a `PARTICIPANT-IN JOBS`.

## 5.1 Reasoning with OilEd

Support for the user is crucial during processes such as editing or ontology refinement as described in [19]. In particular, we can make use of a *reasoner* such as FaCT [13] to assist in the process. Tasks that can benefit from reasoning include:

- checking the consistency of concept definitions;
- producing an inferred concept hierarchy based on the definitions given.

The first of these can prove particularly useful when the ontology has been richly axiomatised.

OilEd interacts with a Description Logic reasoner, allowing a modeller to perform the tasks described above. Reasoning is currently carried out as a “batch” process (again see [3] for details), with the modeller explicitly requesting that the model be checked for consistency and inferred subsumption relations. The OWL ontology is translated to an equivalent *SHOIQD* model and transmitted to a reasoner. The knowledge base is classified, and satisfiability questions can now be asked for each of the basic concepts in the model. In addition, the tool can query for the concept hierarchy induced by the axioms in the model.

Any concept definitions which have been found to be inconsistent are then reported to the user (for example in Figure 3 the concept `EMPLOYEES` has been found to be inconsistent).

## 5.2 Reasoner Integration (pre-KAON)

Integration of OilEd with FaCT pre-KAON is through the use of the `dig.Reasoner` interface as described in Section 4. The user is offered two choices when connecting to a reasoner:

- Connect using a *Local* Reasoner.
- Connect using an *HTTP* Reasoner.

Depending on the choice the user makes, the application creates an appropriate object to handle the reasoner requests. If *Local* is selected, a FaCT reasoner is started on the local machine and a `dig.FaCTReasoner` object is created to process any reasoning requests via that reasoner. Communication between the client and reasoner (i.e. the passing of requests and responses) is done “in-memory”. If *HTTP* is selected, a `dig.HTTPReasoner` is created which passes requests off to a given URL.

The benefits of using this approach (even without the use of the KAON Server as discussed later) are clear. The client application is now isolated from the particular implementation details of the reasoner. In particular, this allowed the integration of OilEd with alternative reasoners to FaCT such as RACER, achieving one of the original aims of the DIG specification, i.e. the seamless integration of alternative reasoning engines.

## 6 Integrating DIG and KAON

Integration of a DIG reasoner requires us to turn the DIG reasoner into a KAON component within the KAON Server. This involves the implementation of a JMX wrapper `kaon.DIGComponent` that serves two purposes:

- making the DIG reasoner a manageable component within the KAON server;
- providing access to the DIG reasoner component.

The JMX wrapper simply wraps the `dig.HTTPReasoner` interface, adding additional functionality required by the KAON Server for managing the component. The further extended architecture is shown in Figure 4. A client library is provided including a `kaon.RemoteReasoner` class that implements the `dig.Reasoner` interface, through communication with a proxy object supplied by the KAON server. This proxy itself makes use of a `dig.Reasoner` in order to supply the required functionality.

## 7 Integrating OilEd and DIG via KAON

Integrating OilEd with a DIG reasoner via the KAON server is now simply a case of allowing the user a third option when connection to the reasoner is requested:

- Connect to a reasoner hosted by the KAON Server.

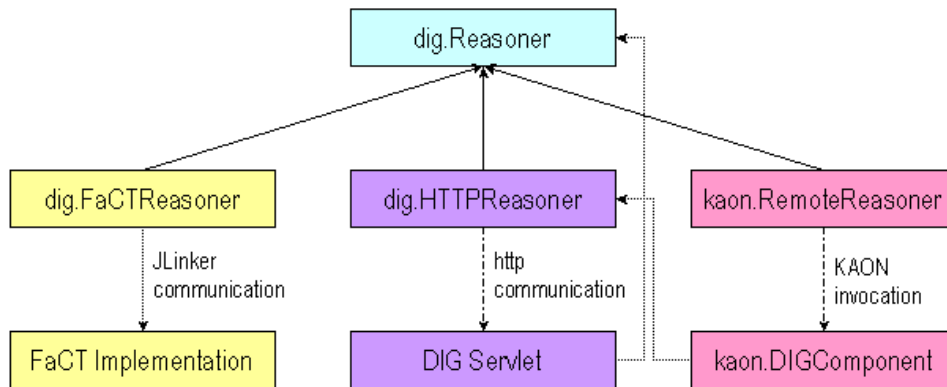


Figure 4: Architecture extended with KAON component

If this option is selected, the client application contacts the KAON Server to request any appropriate components that the server may host. This is currently achieved through requesting a known named component, but in the future we would expect to be able to use the Registry functionality of the Server to provide information about available reasoners. A `kaon.RemoteReasoner` object (see Section 6) is created by the client and this is used to pass requests to the reasoning component hosted by the server.

The only alteration needed to the existing client code is in the creation of the proxy object `kaon.RemoteReasoner`, which uses standard calls from the client library provided for the KAON server. Thus the impact on the client side development is minimal.

Note that in this integration, the functionality of the reasoner is effectively opaque to the KAON Server – the server merely passes requests to and from the reasoner without any real knowledge of the structure or format of the requests and responses. Some benefits can be gained by through the use of the KAON Server:

- The client need not worry about management of reasoners. For example, the server could choose to bring reasoning services up and down as required.
- Through use of the registry, the client can seek reasoners that provide a particular class of service, for example supporting reasoning with concrete data types. Additional metadata may be available concerning the reasoner such as performance figures, which could assist in deciding which of a number of alternative reasoners to select.

## 8 Integrating OilEd and an RDF store via KAON

OilEd uses a bespoke data structure to represent DAML+OIL and OWL models. Functionality to support import and export of these data structures to other concrete representations, in particular those based on RDF/XML, is also provided. In the standard, pre-KAON tool, this import and export is performed on files. Thus a user can create a

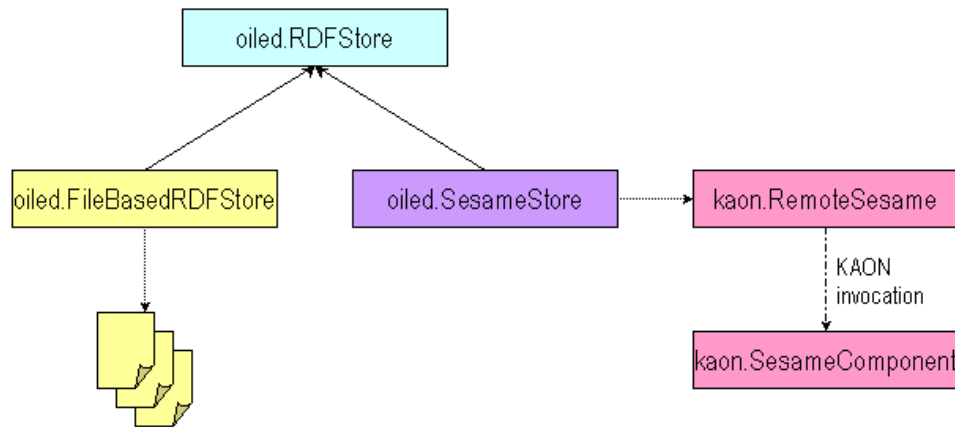


Figure 5: Architecture for RDF storage

new ontology, add a number of classes and properties along with their definitions, and then save the resulting ontology as an OWL-RDF file.

Within WonderWeb, the KAON Server can provide access to an RDF or Triple store [18]. This can then be used to store the RDF models rather than writing to local files. Benefits here are the use of a network-available resource, facilitating the sharing of models and the ability for users to access their models from diverse locations. In addition, this also provides access to additional functionality that may be provided by the RDF storage component, such as versioning [15].

As with the integration of the DIG reasoner, integrating OilEd with an RDF store component hosted by KAON requires little change to the existing code.

The OilEd code makes use of an interface providing access to RDF storage. This interface provides a somewhat coarse-grained level of functionality, effectively allowing the bulk storage of a “chunk” of RDF, and the retrieval of a “chunk” of RDF. In the pre-KAON implementation, as described above, the implementation provided simply loads and stores the RDF in local files.

Models are loaded and saved on user request. We have *not* considered integration with the RDF store at a finer granularity (for example, making changes to the RDF representation as the user edits the model). This is due primarily to the problems of reconciling the triple-based representation of RDF with the higher-level abstract structures that are present in OWL and DAML+OIL models. These issues are discussed at some length in [2] and [8].

Integration of a KAON hosted component for RDF storage simply boils down to providing client code that backs the RDF store interface with an implementation using the provided client API for the KAON server triple store component [18]. Code is also needed to create the proxy – as discussed in Section 7 this uses standard calls from the KAON client API. A simple architecture diagram is shown in Figure 5.

## 9 Conclusions

This document described how an existing tool, OilEd and a reasoner, FaCT, are integrated into the KAON server as a client and server component respectively.

## References

- [1] S. K. Bechhofer, C. A. Goble, A. L. Rector, W. D. Solomon, and W. A. Nowlan. Terminologies and Terminology Servers for Information Environments. In *Eighth International Workshop on Software Technology and Engineering Practice – STEP97*, pages 484 – 497, London, UK, 1997. IEEE Computer Society.
- [2] Sean Bechhofer, Carole Goble, and Ian Horrocks. DAML+OIL is not enough. In *SWWS-1, Semantic Web working symposium*, Jul/Aug 2001.
- [3] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a Reasonable Ontology Editor for the Semantic Web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, volume 2174 of *LNAI*, pages 396–408, Vienna, Sep 2001. Springer-Verlage.
- [4] Sean Bechhofer, Ian Horrocks, Carole A. Goble, and Robert Stevens. Oiled: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence*, volume 2174 of *LNAI*, pages 396–408, Vienna, September 19-21 2001. Springer-Verlag.
- [5] Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider, and Sergio Tessaris. A Proposal for a Description Logic Interface. In *DL99, International Workshop on Description Logics*, Linköping, Sweden, 1999.
- [6] Sean Bechhofer, Ian Horrocks, Peter F. Patel-Schneider, and Sergio Tessaris. A Proposal for a Description Logic Interface. In Lambrix et al. [16].
- [7] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG Description Logic Interface. In *DL'03 International Workshop on Description Logics*, Rome, Italy, 2003.
- [8] Sean Bechhofer, Raphael Volz, and Phillip Lord. Cooking the Semantic Web with the OWL API. In *To Appear in: ISWC Second International Semantic Web Conference*, Sanibel Island, Florida, October 2003.
- [9] Ian Horrocks Dmitry Tsarkov. Reasoner Prototype. WonderWeb Project Deliverable 13, WonderWeb, 2003.
- [10] Enrico Franconi and Gary Ng. The i.com Tool for Intelligent Conceptual Modelling. In *7th Intl. Workshop on Knowledge Representation meets Databases (KRDB'00)*, Berlin, Germany, August 2000.

- [11] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and Berners-Lee T. Hypertext Transfer Protocol – HTTP/1.1. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [12] Volker Haarslev and Ralf Möller. Description of the RACER System and its Applications. In *Proceedings of the International Workshop on Description Logics (DL-2001)*, Stanford, USA, August 2001.
- [13] I. Horrocks. The FaCT system. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, number 1397 in Lecture Notes in Artificial Intelligence, pages 307–312. Springer-Verlag, Berlin, May 1998.
- [14] I. Horrocks. FaCT and iFaCT. In Lambrix et al. [16], pages 133–135.
- [15] Michel Klein. Versioning of Distributed Ontologies. WonderWeb Project Deliverable 13, WonderWeb, 2003.
- [16] P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors. *Proceedings of the International Workshop on Description Logics (DL'99)*, 1999.
- [17] Peter F. Patel-Schneider and Bill Swartout. Description logic specification from the KRSS effort, 1993.
- [18] Raphael Volz, Daniel Oberle, Steffen Staab, and Rudi Studer. Triple Client. WonderWeb Project Deliverable 8, WonderWeb, 2003.
- [19] WonderWeb Project. A Case Study in Semantic Web Engineering. Review Scenario Paper.
- [20] World Wide Web Consortium. XML Schema. <http://www.w3.org/XML/Schema>.