

Software Tools for Ontology Design and Maintenance

Deliverable TONES-D15

Diego Calvanese¹, Bernardo Cuenca Grau³, Enrico Franconi¹, Ian Horrocks³,
Alissa Kaplunova⁵, Carsten Lutz⁴, Ralf Möller⁵, Baris Sertkaya⁴,
Sergio Tessaris¹, Anni-Yasmin Turhan⁴

¹ Free University of Bozen-Bolzano

² Università di Roma “La Sapienza”

³ The University of Manchester

⁴ Technische Universität Dresden

⁵ Technische Universität Hamburg-Harburg



Project:	FP6-7603 – Thinking ONtologies (TONES)
Workpackage:	WP3 – Tasks for Ontology Design and Maintenance
Lead Participant:	TU Dresden
Reviewer:	—
Document Type:	Deliverable
Classification:	Consortium Access Only
Distribution:	TONES Consortium
Status:	Final
Document file:	D15_ToolsDesign.pdf
Version:	1.1
Date:	March 31, 2007
Number of pages:	55

Document Change Record		
Version	Date	Reason for Change
v.0.1	March 1, 2007	Outline
v.1.0	March 19, 2007	First version
v.1.1	March 31, 2007	Final version

Contents

1	Introduction	4
2	RacerPro	5
2.1	Introduction	5
2.2	Interfaces	6
2.3	Specific Extensions	6
2.4	New Services for Ontology Design and Maintenance	8
2.5	Optimizations	9
3	FaCT++	9
3.1	Introduction	9
3.2	FaCT++ Optimisations	10
3.2.1	Preprocessing Optimisations	10
3.2.2	Satisfiability Checking Optimisations	11
3.2.3	Classification Optimisations	12
3.3	How to Use	12
3.3.1	Installation	13
3.3.2	Usage	13
4	CEL	14
4.1	Introduction	14
4.2	Optimizations	16
4.3	How to use	18
5	Swoop	22
5.1	Introduction	22
5.2	Swoop Features	22
5.2.1	Ontologies based on the Web Architecture	22
5.2.2	Editing Web Ontologies	23
5.2.3	Adhering to OWL Specifications: Presentation and Reasoning	25
5.2.4	Reasoning in OWL	26
5.2.5	Ontology Debugging and Repair	26
5.3	How to Use	27
6	RacerPorter	28
6.1	Introduction	28
6.2	Towards User-Friendly and Scalable OBITS	29
6.3	RACERPORTER – How to Use	32
6.4	Some Notes About Performance	35
6.5	Conclusion	35

7	iCom	35
7.1	Introduction	35
7.2	Optimisations	37
7.3	How to use	37
8	OntoExtract	39
8.1	Introduction	39
8.2	How to use	39
8.2.1	Input file format	39
9	SONIC	40
9.1	Introduction	40
9.2	Optimizations	42
9.3	How to use	43
10	InstExp	45
10.1	Introduction	45
10.2	Optimizations	45
10.3	Usage	45
11	DL²RL	48
11.1	Introduction	48
11.2	DL ² RL – How to Use	49
11.3	Further Work and Optimizations	51

1 Introduction

This document accompanies the software deliverable D15 “Software Tools for Ontology Design and Maintenance” of the TONES project, providing an overview of the delivered software packages along with some basic information on how to install and run them. The delivered software tools are implementations of the techniques described in deliverable D13 [ton07], usually enriched with optimization techniques in order to make them more efficient. Roughly, the tools can be divided into four groups:

1. Classical reasoners: RacerPro, FaCT++, and CEL.

The main strength of these tools is in classical reasoning services for ontologies, namely in checking consistency and computing the subsumption hierarchy of the concepts defined in the ontology. However, they also implement a considerable number of additional services such as error management and knowledge base querying.

2. Frontends: Swoop, iCom, and RacerPorter

Swoop and iCom are user frontends for ontology design and editing. Swoop provides an interface reminiscent of frame systems, which allows to browse the class hierarchy and to view and edit all relevant details of the classes of an ontology. ICom focusses on the graphical display and editing of ontologies using an extended version of UML class diagrams. Both tools are equipped with a highly configurable API that allows them to interact with different reasoners. RacerPorter is a tool to be used with RacerPro. It allows to visualize an ontology and display statistical information about it.

3. Database-related tools: iCom and OntoExtract

The interplay between ontologies and databases is one of the focusses of the TONES project. The iCom tool addresses this subject by providing a unifying tool for editing ontologies and conceptual database schemas, thus facilitating the integrated design of ontologies and databases. Complementing this approach, the OntoExtract tool allows to automatically extract initial ontologies from existing database schemas, which can then be manually fine-tuned.

4. Reasoners for novel reasoning services: SONIC, INSTEXP, and DL²RL

These tools implement reasoning services for ontologies that have only been proposed recently. The SONIC tool provides a wealth of such services, addressing in particular the automatic generation of concepts. The INSTEXP tool supports and guides the semi-automatic completion of ontologies by adding either new subsumption relationships or new counterexamples. The DL²RL tool allows to construct models for ontologies that can then be inspected by the designer to check whether the ontology under development describes the intended structures.

The central part of this deliverable is organized as follows. There is one section for each tool, which is structured into (i) an introductory part, (ii) a discussion of optimization techniques used in the tool, and (iii) a section describing how to use the tool. In (i),

we give a general overview of the tool and explain which ontology design and maintenance tasks it addresses. If the tool was not developed completely within the TONES project (e.g. because its development started already before the project), we point out the concrete contributions that have been made within TONES. For technical details of the implemented algorithms, we refer to deliverable D13 [ton07]. In (ii), we briefly discuss optimization techniques and heuristics that have been used to make the implemented algorithms more efficient. We point out that for some tools which are in an early stage of development, finding appropriate optimizations is, as of now, future work. In (iii), we give brief instructions on how to install the tools, the system requirements, and which additional software components are required. If possible, we also include some basic information on how to configure and use the tool. However, note that this section is not intended to be a full-fledged manual. Whenever available, a manual is included on the CD as part of the software package.

We point out that many of the tools described in this document are multi-purpose, and not limited to providing support for ontology design and maintenance, only. For example, RacerPro comprises a sophisticated querying engine and thus can also be used when deploying an ontology in an application. Similarly, the Swoop tool supports the integration and interoperation of ontologies. Due to this generality, many tools included in the current software deliverable are actually also deliverables for other workpackages. In this document, we focus on describing the relevance of the delivered tools for ontology design and maintenance. The contributions of the delivered tools to other workpackages will be the subject of later documents.

We also remark that not all of the techniques reported about in deliverable D13 have made their way into tools. The reason is that some of the techniques, such as deciding conservative extensions, are in a rather early state of investigation in which decidability and complexity results have been obtained, but the existing algorithms do not appear to be well-suited for implementation. In these cases, further research into practicable algorithms and/or optimization techniques are required before even experimental implementations are worthwhile to pursue.

A final issue is the interplay of the presented tools. Whenever possible, we have tried to use common interfaces and representation standards in order to facilitate a tight coupling. For example, the TONES consortium has been one of the main driving forces behind the advancement of the DIG standard, which describes the most common API for ontology-processing tools, to its current version 2.0. Due to these efforts, frontends such as Swoop and iCom can be effortlessly combined with different reasoning backends such as RacerPro, FaCT++, CEL, and SONIC. It will be part of the final TONES demo to practically illustrate the interplay of the tools.

2 RacerPro

2.1 Introduction

RACERPRO [HM01a] is under continuous development since 1998 (commercial support is available for two years now). The system is used for ontology design and maintenance

(offline usage of ontologies) as well as for using ontologies in running applications that rely on reasoning (online usage of ontologies). Since ontologies get larger and larger, and new application fields use ontologies these days, the demands on system architecture ever increase.

Basically, the system implements the description logic SHIQ(Dn) with TBoxes and ABoxes (see [ton07] for details about syntax and semantic of description logics). All standard DL inference services for ontology design and maintenance are provided by RACERPRO. In order to assist the creation of practical applications, the RACERPRO system includes several extensions the development of which has been partially supported by TONES project (we indicate this with * in the following text).

2.2 Interfaces

Several interfaces are available for RACERPRO. As usual, the reasoner supports file-based interaction as well as socket-based communication with end-user applications or graphical interfaces for ontology development and maintenance. Input can be specified in various syntaxes, e.g., KRSS (TCP), DIG 1.1 (HTTP), or OWL DL (HTTP). A parser for DIG 2.0 [Sea06] is in preparation. As an extension to DIG 1.1, RACERPRO already supports an XML-based interface for conjunctive queries. The specification of this interface is also proposed as part of DIG 2.0 with some slight modifications [Ali06]. The RACERPRO implementation of DIG 2.0 will support also expressive constraints (see the subsection about concrete domains presented below). Unparsers from the internal meta model to a textual representation of ontologies are available for all syntaxes.

In particular, for DIG 2.0 it will be the case that not all syntactic constructs might be implemented by a certain reasoner. For instance, DIG 2.0 includes nominals as part of the TBox (this also holds for OWL DL). Currently, RACERPRO fully supports nominals as part of ABoxes. Nominals in the TBox are approximated by concept names. For fully supporting the OWL 1.1. fragment of DIG 2.0, also acyclic role axioms have to be provided by the RACERPRO implementation. It is well known, however, that for some purposes, even DIG 2.0 is not expressive enough. Further extensions are required that we describe in the next section.

2.3 Specific Extensions

Rules applied to ABox individuals* Rule specifications are well known (e.g., from the W3C SWRL specification [SWR04]), but different systems support different semantics (for details of the RACERPRO semantics for rules, see the RACERPRO reference manual [Rac]). In RACERPRO, rules can be seen as a convenient specification about how to extend the set of assertions in an ABox. In addition, rules can be used as named queries that can be reused in other queries. Rule design is also part of ontology design. Rule bodies can be checked for subsumption (grounded semantics). Rules in RACERPRO can be specified with a KRSS or SWRL syntax.

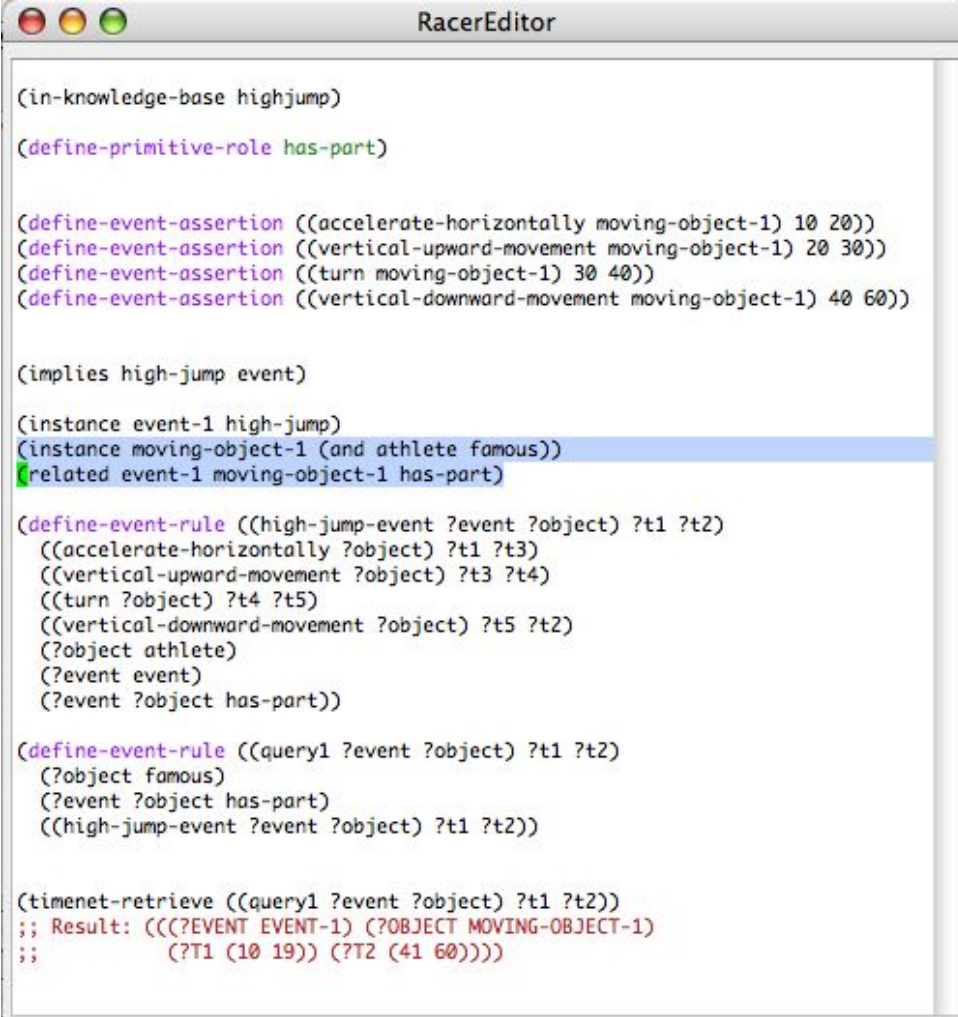
Concrete domains In some sense, OWL is rather inexpressive in that it does not support constraints between attribute values of different individuals. For instance, in

OWL it is not possible to state that Mike's brother, called John, is ten years older than Mike, and Mike is a car driver (and the ontology says that car drivers must be older than 18). Does this mean that, concerning the age, John is allowed to drive a car as well? RACERPRO supports inequations about linear polynomials over the reals and over positive integers. In addition, RACERPRO allows for expressing min/max restrictions over integers as well as (in)equalities over strings. If individuals are part of the ontology (and OWL even supports nominals in the TBox), consistency checking is an important issue at ontology-development time. At the time of this writing, constraints between different individuals are still not supported by the latest proposal for the new OWL language: OWL 1.1 [Ber06]. They are supported by DIG 2.0, however.

Support for spatial reasoning* As a generalization of concrete domains it is possible to associate graph-based representation formalisms with an ABox. Individuals in the ABox are associated with nodes in the graph in a bidirectional way. An associated graph is called a substrate (on which abstract knowledge in the ABox is built). An example for a substrate can be a spatial representation formalism. Nodes correspond to spatial objects, and edges in the graph correspond to spatial relations (e.g., topological relations such as in the RCC formalism). Depending on the semantics of the substrate, i.e., depending on the semantics of the spatial relations, reasoning services are provided. The vocabulary declared for denoting nodes and edges is available also in the query language for the ABox, and the substrate reasoning services are employed for ABox query answering. With a spatial substrate used in the query language, it is possible to find individuals that are associated with spatial objects that satisfy certain spatial restrictions (quantitative or qualitative) at the substrate level as well as conceptual and relational restriction at the ABox level. The combination of spatial and ontological reasoning is provided by the query language. Note that, for instance, the semantics of spatial relations as defined in the Region Connection Calculus (RCC) cannot be obtained by using role axioms as offered in OWL 1.1 because the role axioms required do not satisfy the acyclicity condition.

Support for temporal reasoning in the context of event recognition* In some applications, temporal aspects have to be handled. For instance, temporal events have to be recognized based on ABox assertions associated with time intervals (temporal propositions). RACERPRO supports rules with time intervals for the definition of event models. In the query language, temporal as well as ontological aspects are combined. Rules with time variables can also be used to compute all events that hold w.r.t. a given ABox and set of temporal propositions.

In Figure 1 an example for a definition of an event together with temporal propositions as well as an ABox and a TBox are given. For the query at the bottom the result is printed in the editor here (for demonstration purposes). The result specifies binding for query variables as well as intervals (lower bound and upper bound specifications) for the start and end timepoints, respectively.



```

(in-knowledge-base highjump)

(define-primitive-role has-part)

(define-event-assertion ((accelerate-horizontally moving-object-1) 10 20))
(define-event-assertion ((vertical-upward-movement moving-object-1) 20 30))
(define-event-assertion ((turn moving-object-1) 30 40))
(define-event-assertion ((vertical-downward-movement moving-object-1) 40 60))

(implies high-jump event)

(instance event-1 high-jump)
(instance moving-object-1 (and athlete famous))
(related event-1 moving-object-1 has-part)

(define-event-rule ((high-jump-event ?event ?object) ?t1 ?t2)
  ((accelerate-horizontally ?object) ?t1 ?t3)
  ((vertical-upward-movement ?object) ?t3 ?t4)
  ((turn ?object) ?t4 ?t5)
  ((vertical-downward-movement ?object) ?t5 ?t2)
  (?object athlete)
  (?event event)
  (?event ?object has-part))

(define-event-rule ((query1 ?event ?object) ?t1 ?t2)
  (?object famous)
  (?event ?object has-part)
  ((high-jump-event ?event ?object) ?t1 ?t2))

(timenet-retrieve ((query1 ?event ?object) ?t1 ?t2))
;; Result: (((?EVENT EVENT-1) (?OBJECT MOVING-OBJECT-1)
;;           (?T1 (10 19)) (?T2 (41 60))))

```

RacerPro 1.9.1 running on localhost:8088 (case: Upper Case) -**- highjump3.race

Figure 1: Event recognition example.

2.4 New Services for Ontology Design and Maintenance

An ontology comprises knowledge about individuals (e.g., as part of the ABox). At ontology-development time a developer might be interested in knowing whether there exists a constellation of individuals in the ABox that satisfy certain conditions. The conditions can be stated as a conjunctive query. If, for whatever reason, the conditions are not satisfied, and corresponding individuals do not exist, or the conditions on the existing individuals are not satisfied, the *abduction reasoning service* can be employed. The abduction service returns proposals what could be added to the ABox in order to satisfy the query. This service is particularly useful for bottom-up ontology construction and can be combined with generalization inference services also investigated in the TONES project.

RACERPRO now also supports a pinpointing inference services for computing TBox axioms as well as ABox axioms that are culprits for an ABox unsatisfiability condition. The integration into RACERPORTER (see below) is under development.

The RACERPRO reasoning engine support cancellation of requests and also considers timeout specifications. For the interaction with other modules, for instance, graphical interfaces such as RACERPORTER, support for progress indication is in preparation.

2.5 Optimizations

For ontology design and maintenance, the following optimizations are important.

- Incremental constraint checking during tableau proof (e.g., for the string domain).
- Integration of optimizations for concrete domain reasoning for OWL datatypes (less expressive than full concrete domains).
- Techniques for using the dependency-tracking mechanism of the reasoner to support explanation generation (glass box approach).
- Optimizations for spatial substrates: spatial indexing, RCC reasoning.

3 FaCT++

3.1 Introduction

FaCT++ is a sound and complete DL reasoner designed as a platform for experimenting with new tableaux algorithms and optimisation techniques.¹ It incorporates most of the standard optimisation techniques, but also employs many novel ones.

DL systems take as input a knowledge base (equivalently an ontology) consisting of a set of axioms describing constraints on the conceptual schema (often called the TBox) and a set of axioms describing some particular situation (often called the ABox). They are then able to answer both “intensional” queries (e.g., regarding concept satisfiability and subsumption) and “extensional” queries (e.g., retrieving the instances of a given concept) w.r.t. the input knowledge base (KB). For the expressive DLs implemented in modern systems, these reasoning tasks can all be reduced to checking KB satisfiability.

When reasoning with a KB, FaCT++ proceeds as follows. A first *preprocessing* stage is applied to the KB when it is loaded into reasoner; it is normalised and transformed into an internal representation. During this process several optimisations (that can be viewed as a syntactic re-writings) are applied.

The reasoner then performs *classification*, i.e., computes and caches the subsumption partial ordering (taxonomy) of named concepts. Several optimisations are applied here, mainly involving choosing the order in which concepts are processed so as to reduce the number of subsumption tests performed.

¹FaCT++ is available at <http://owl.man.ac.uk/factplusplus>.

The classifier uses a KB *satisfiability* checker in order to decide subsumption problems for given pairs of concepts. This is the core component of the system, and the most highly optimised one.

FaCT++ can be downloaded at the following address: <http://owl.man.ac.uk/factplusplus/>. Within TONES FaCT++ has been extended with new optimization techniques and to support *SRIOQ*, the logic underlying OWL 1.1.

3.2 FaCT++ Optimisations

3.2.1 Preprocessing Optimisations

Lexical normalisation and *simplification* is a standard rewriting optimisation primarily designed to promote early clash (inconsistency) detection, although it can also simplify concepts and even detect relatively trivial inconsistencies. The basic idea is that all concepts are transformed into a *simplified normal form* (SNF), where the only operators allowed in SNF are negation (\neg), conjunction (\sqcap), universal restriction (\forall) and (qualified) at-most restriction (\leq). In FaCT++, the translation into SNF is performed on the fly, during the parsing process. At the same time, some simplifications are applied to concept expressions, including constant elimination (e.g., $C \sqcap \perp \rightarrow \perp$), expression elimination (e.g., $\neg\neg C \rightarrow C$), and subsumer elimination (e.g., $C \sqcap D \rightarrow C$ for D a known subsumer of C).

Absorption is a widely used rewriting optimisation that tries to eliminate General Concept Inclusion axioms (GCIs, axioms in the form $C \sqsubseteq D$, where both C and D are complex concept expressions), as GCIs left in the TBox invariably lead to a significant decrease in the performance of tableaux based satisfiability/subsumption testing procedures. In FaCT++, GCIs are eliminated by absorbing them into either concept definition axioms (*concept absorption*) or role domain axioms (*role absorption*). Role absorption is particularly beneficial from the point of view of the CD-classification optimisation (see Section 3.2.3), as it eliminates GCIs without reducing the number of concepts to which CD-classification can be applied.

Told Cycle Elimination is a technique that we assume is used in most modern reasoners, although we know of no reference to it in the literature. Definitional cycles in the TBox can lead to several problems, and in particular cause problems for algorithms that exploit the told subsumer hierarchy (see Section 3.2.3). These cycles are, however, often quite easy to eliminate. Assume, for example, that $A_1 \dots A_n$ are concept names, $C_1 \dots C_n$ are arbitrary concept expressions, and \bowtie is either \sqsubseteq or \doteq . The axioms $A_1 \bowtie A_2 \sqcap C_2, A_2 \bowtie A_3 \sqcap C_3, \dots, A_n \bowtie A_1 \sqcap C_1$ include a definitional cycle, because the r.h.s. of the first axiom (indirectly) refers to the name on its l.h.s. The cycle can, however, be eliminated by transforming the axioms into $A_2 \doteq A_1, \dots, A_n \doteq A_1, A_1 \sqsubseteq C_1 \sqcap C_2 \dots \sqcap C_n$.

Synonym Replacement is used to extend simplification possibilities and improve early clash detection. If the only axiom with C on the left hand side is $C \doteq D$, then C is called a *synonym* of D . For a set of concept names, all of which are synonymous, FaCT++ uses a single “canonical” name in all concept expressions in the KB.

FaCT++ first translates all input concepts into SNF, with subsequent transformations being designed to preserve this form. After simplification and absorption, FaCT++ re-

peatedly performs cycle and synonym elimination steps until there are no further changes to the KB.

3.2.2 Satisfiability Checking Optimisations

The FaCT++ system was designed with the intention of implementing DLs that include inverse roles, and of investigating new optimisation techniques, including new ordering heuristics. In order to deal more easily with inverse roles, and to allow for more flexible ordering of the tableaux expansion, FaCT++ uses a *ToDo list*, instead of the usual top-down approach, to control the application of the expansion rules. The basic idea behind this approach is that rules may become applicable whenever a concept is added to a node label. When this happens, the relevant node/concept pair is added to the *ToDo list*. The *ToDo list* sorts entries according to some order, and gives access to the “first” element in the list. The tableaux algorithm repeatedly removes and processes list entries until either a clash occurs or the list becomes empty.

Dependency-directed backtracking (Backjumping) is a crucial and widely used optimisation. Each concept in a completion tree label is labelled with a *dependency set* containing information about the branching decisions on which it depends. In case of a clash, the system backtracks to the most recent branching point where an alternative choice might eliminate the cause of the clash.

Boolean constant propagation (BCP) is another widely used optimisation. As well as the standard tableau expansion rules, additional inference rules can be applied to the formulae occurring in a node label, usually with the objective of simplifying them and reducing the number of nondeterministic rule applications. BCP is probably the most commonly used simplification, the basic idea being to apply the inference rule

$$\frac{\neg C_1, \dots, \neg C_n, C_1 \sqcup \dots \sqcup C_n \sqcup C}{C}$$

to concepts in a node labels.

Semantic Branching is another rewriting optimisation, the idea being to rewrite disjunctions of the form $C \sqcup D$ as $C \sqcup (\neg C \sqcap D)$. If choosing C leads to clash, then the $\neg C$ in the second disjunct (along with BCP) ensures that C will not be added to the node label again by some other nondeterministic expansion.

Ordering Heuristics can be very effective, and have been extensively investigated in FaCT++ [TH05]. Changing the order in which nondeterministic expansions are explored can result in huge (up to several orders of magnitude) differences in reasoning performance. Heuristics can be used to choose a “good” order in which to try the different possible expansions. In practise, this usually means using heuristics to select the way in which expansion rules are applied to the disjunctive concepts in a node label, with a heuristic function being used to compute the relative “goodness” of each candidate expansion.

Heuristics may select an expansion-ordering based on, e.g., (ascending or descending order of) concept size, maximum quantifier depth, or frequency of usage. In order to reduce the cost of computing the heuristic function, FaCT++ computes and caches relevant values for each concept as the KB is loaded. As no one heuristic performs well in all cases, FaCT++ also selects the heuristics to be used based on an analysis of the structure of the input KB.

3.2.3 Classification Optimisations

As mentioned above, the focus here is on reducing the number of subsumption tests performed during classification. In FaCT++, this is achieved by both reducing the number of comparisons and by substituting cheaper (but incomplete) comparisons where possible.

Definitional Ordering is a well known technique that uses the syntactic structure of TBox axioms to optimise the order in which the taxonomy is computed. E.g., given an axiom $C \sqsubseteq D$, with C a concept name, FaCT++ will delay adding C to the taxonomy until all of the concepts occurring in D have been classified. In some cases this technique allows the taxonomy to be computed “top down”, thereby avoiding the need to check for subsumees of newly added concepts.

Similarly, the structure of TBox axioms can be used to avoid (potentially) expensive subsumption tests by computing a set of (trivially obvious) *told subsumers* and *told disjoint*s of a concept C . E.g., if the TBox contains an axiom $C \sqsubseteq D_1 \sqcap D_2$, then FaCT++ treats both D_1 and D_2 , as well as all *their* told subsumers, as told subsumers of C , and if the TBox contains an axiom $C \sqsubseteq \neg D \sqcap \dots$, then D is treated as a told disjoint of C . The classification algorithm can then exploit obvious (non-) subsumptions between concepts and their told subsumers (disjoints).

Model Merging is a widely used technique that exploits cached partial models in order to perform a relatively cheap but incomplete non-subsumption test. If the cached models for D and $\neg C$ can be merged to give a model of $D \sqcap \neg C$, then the subsumption $C \sqsubseteq D$ clearly does not hold.

Completely Defined Concepts is a novel technique used in FaCT++ to deal more effectively with wide (and shallow) taxonomies. In this case, some concepts in the taxonomy may have very many direct subsumees, rendering classification ordering optimisations ineffective. It is often possible, however, to identify a significant subset of concepts whose subsumption relationships are completely defined by told subsumptions. FaCT++ computes a taxonomy for these concepts without performing any subsumption tests.

Clustering is another technique that addresses the same problem [HM01b]. The idea here is to introduce new “virtual concepts” into the taxonomy in order to produce a deeper and more uniform structure. These concepts are asserted to be equivalent to the union of a number of sibling concepts and are inserted in the taxonomy in between these concepts and their common parent.

3.3 How to Use

FaCT++ currently supports the *SRIOQ* description Logic language, which corresponds to the OWL 1.1 ontology language. The current version is 1.1.5. This is source distribution package so it can be used on different platforms. It was tested on Windows, Linux and MacOS X.

FaCT++ is distributed under GNU Public License (GPL). Full text of license can be found at <http://www.gnu.org/licenses/gpl.txt>.

3.3.1 Installation

For building system you will need GNU c++ compiler and GNU make (version 3.3 and higher were tested). Change GENERAL_DEFINES macro in src/Makefile.include to make it suitable for your computer. Then just run "make".

In order to compile DIG part you will also need an XML parsing library Xerces-c (freely available at <http://xml.apache.org/xerces-c/>). Make sure that Xerces-c package is installed system-wide or you have environment variable XERCESCROOT which points to Xerces-c root directory.

In order to compile OWL-API interface (src/FaCTPlusPlusJNI/) it is necessary to have JNI development files (jni.h) available.

3.3.2 Usage

The Models.lisp directory of this distribution contains some files that support FaCT++ reasoning as well as examples of KBs.

To use standalone reasoner user should usually perform the following steps:

- create an ontology using the FaCT++ input language;
- create a working directory (i.e. TEST) for FaCT++ using the command create-new-test TEST ontology; where "ontology" is the name of the file containing your FaCT++ ontology
- inside TEST directory run "make".

This will run FaCT++ reasoner on the newly created config-file for the given ontology. The results of FaCT++ reasoning appear in following files:

- Taxonomy.Roles contains information about the roles taxonomy;
- Taxonomy.log contains information about the concept taxonomy (if it was requested);
- dl.res contains full information about the ontology and some statistical information about the reasoning process.

Concerning ontology creation, there are three ways of creating an ontology for FaCT++:

- hand-made ontology. This way is not recommended for the end user;
- using OilEd (<http://oiled.man.ac.uk>). Load an ontology to the OilEd then choose ExportFaCT++ lisp;
- from the OWL source using the OWL Ontology Converter; (<http://phoebus.cs.man.ac.uk:9999/OWL/Converter>). Set the ontology URL to the OWL ontology, choose FaCT++ as the output language, press Convert and then copy the resulting ontology text to the FaCT++ ontology file.

There are a number of options that could influence the reasoning process. All options, their format and description are given in the config file, which is generated by the create-new-test script.

- FaCT++ as an HTTP DIG reasoner: Run FaCT++ server with optional parameter “-port {port}”. Default value of “port” is 3490;
- FaCT++ as an HTTP OWL reasoner: Use FaCT++ as in the case of DIG. Then, connect your OWL editor like Protege (<http://protege.stanford.edu/>) to FaCT++.

4 CEL

4.1 Introduction

The system CEL is a first step towards realizing the dream of a description logic system that offers both sound and complete polynomial-time algorithms and expressive means that allow its use in real-world applications. It is based on recent theoretical advances that have shown that the description logic (DL) \mathcal{EL} , which allows for conjunction and existential restrictions, and some of its extensions have a polynomial-time subsumption problem even in the presence of concept definitions and so-called general concept inclusions (GCI) [BBL05]. The DL \mathcal{EL}^+ handled by CEL extends \mathcal{EL} by so-called role inclusions (RI). On the practical side, it has turned out that the expressive power of \mathcal{EL}^+ is sufficient to express several large life science ontologies. In particular, the Systematized Nomenclature of Medicine (SNOMED) [CRP⁺93] employs \mathcal{EL} with RIs and acyclic concept definitions. The Gene Ontology (GO) [The00] can also be expressed in \mathcal{EL} with acyclic concept definitions and one transitive role (which is a special case of an RI). Finally, large parts of the Galen Medical Knowledge Base (GALEN) [RH97] can be expressed in \mathcal{EL} with GCIs and RIs.

For the complete syntax and semantics of the DL \mathcal{EL} and relevant extensions thereof, we refer to Section 4 of Deliverable D13. There, you will also find additional constructs which are not implemented yet in CEL. To make this section self-contained, however, we briefly mention the syntax elements, and illustrate their use by a small example. Like in other DLs, \mathcal{EL}^+ *concepts* are inductively defined starting with the sets of *concept names* N_C and *role names* N_R .² Each concept name A is a concept, and so are the *top concept* \top , *conjunction* $C \sqcap D$, and *existential restriction* $\exists r.C$. An \mathcal{EL}^+ *ontology* is a finite set of *general concept inclusions (GCI)* of the form $C \sqsubseteq D$ for concepts C, D , and *complex role inclusions (RI)* of the form $r_1 \circ \dots \circ r_n \sqsubseteq s$ for roles r_1, \dots, r_n, s . A *primitive concept definition (PCDef)* $A \sqsubseteq D$ is a GCI with the left-hand side a concept name, while a (non-primitive) *concept definition (CDef)* $A \equiv D$ can be expressed using two GCIs. It is worthwhile to note that RIs generalize at least three expressive means important in bio-medical applications: role hierarchy, transitive role, and so-called right-identity axioms [CRP⁺93]. One of the most prominent inference problems for DL ontologies is

²Unlike some reasoners, CEL does not presume these sets of names to be disjoint, hence name punning in an ontology is possible.

Endocardium	\sqsubseteq	Tissue	\sqcap	\exists cont-in.HeartWall	\sqcap	\exists cont-in.HeartValve
HeartWall	\sqsubseteq	BodyWall	\sqcap	\exists part-of.Heart		
HeartValve	\sqsubseteq	BodyValve	\sqcap	\exists part-of.Heart		
Endocarditis	\sqsubseteq	Inflammation	\sqcap	\exists has-loc.Endocardium		
Inflammation	\sqsubseteq	Disease	\sqcap	\exists acts-on.Tissue		
HeartDisease	\equiv	Disease	\sqcap	\exists has-loc.Heart		
part-of	\sqsubseteq	cont-in				
has-loc \circ cont-in	\sqsubseteq	has-loc				

Figure 2: An example \mathcal{EL}^+ ontology (motivated by GALEN).

classification: compute the subsumption hierarchy of all concept names occurring in the ontology.

As an example, we consider the \mathcal{EL}^+ ontology in Fig. 2, where all capitalized words are concept names and all lowercase words are role names. This small ontology contains 5 GCIs (which are indeed PCDefs), a CDef, and 2 RIs (more precisely a role hierarchy and a right-identity axiom) expressing a piece of clinical knowledge about *endocarditis* and related concepts and roles. It is not hard – yet also nontrivial – to infer from this ontology that endocarditis is classified as heart disease, i.e., $\text{Endocarditis} \sqsubseteq_{\mathcal{O}} \text{HeartDisease}$. In fact, (i) *Endocarditis* implies *Inflammation* and thus *Disease*, which yields the first conjunct in the definition of *HeartDisease*. Moreover, (ii) $\exists \text{has-loc. Endocardium}$ implies $\exists \text{has-loc. } \exists \text{cont-in. HeartWall}$ and thus $\exists \text{has-loc. } \exists \text{cont-in. } \exists \text{part-of. Heart}$, which, in the presence of both RIs, implies $\exists \text{has-loc. Heart}$, satisfying the second conjunct in the definition of *HeartDisease*.

The development of this lightweight reasoner is partially supported by TONES, as well as by the Germany Research Foundation under grant DFG BA 1122/11-1. The design and development of CEL has been started in the first quarter of 2005 [Sun05b, BLS05, BLS06b, BLS06a], and gradually maintained and enhanced over time. The most remarkable new features³ obtained during the project include the enhanced logical expressive power (A-Box, concept disjointness constraints, domain and range restrictions on roles are now supported), fast computation of subsumption hierarchy, support for DIG 1.1 interface and, most notably, scalability.

The latest and prior distributions of the CEL reasoner can be downloaded from the CEL homepage at <http://lat.inf.tu-dresden.de/systems/cel/>. Installation and system requirements will be mentioned later in Subsection “How to use”. In the following, we describe the novel algorithm used in the CEL reasoner including some related optimization techniques.

³Another important feature not listed above is the non-standard inference for explaining logical consequences (such as subsumption) by means of axiom pinpointing. The pinpointing module for CEL is currently being developed and thus will not be available in the distribution CD. This feature should however be ready for testing in the forthcoming deliverable.

4.2 Optimizations

The implementation of CEL is underlain by the polytime \mathcal{EL}^{++} subsumption algorithm presented in Section 4 of Deliverable 13. Since the reasoner only supports the sublanguage \mathcal{EL}^+ , only the Completion Rules **CR1** – **CR4**, **CR11** and **CR12**) are relevant. For the sake of brevity, the abstract algorithm will not be presented again. Nevertheless, it is important to make a few remarks as follows:

- To reduce the number of *new* concept names introduced during normalization, we adopt a slightly modified normal form in which n -ary conjunction of concept names is allowed on the left-hand side of GCIs, i.e. $A_1 \sqcap \dots \sqcap A_n \sqsubseteq B$.
- **CR1** and **CR2** are generalized to support the new form of GCIs, and henceforth referred to as **R1** as follows:
If $A_1, \dots, A_n \in S(X)$, $A_1 \sqcap \dots \sqcap A_n \sqsubseteq B \in \mathcal{O}$, and $B \notin S(X)$
then $S(X) := S(X) \cup \{B\}$
- In what follows, **CR3**, **CR4**, **CR11** and **CR12** are renamed to **R2** – **R5**, respectively.

One of the main problems to be solved when implementing the rule-based algorithm is to develop a good approach for finding the next completion rule to be applied. If this is realized by a naïve brute-force search, then one cannot expect an acceptable runtime behavior on large inputs. As a solution to this problem, we propose a refined version of the algorithm, which is inspired by the linear-time algorithm for satisfiability of propositional Horn formulas proposed in [DG84]. This version uses a set of queues, one for each concept name appearing in the input ontology, to guide the application of completion rules. Intuitively, the queues list modifications to the data structure (i.e. to the sets $S(A)$ and $R(r)$) that still have to be carried out. The possible entries of the queues are of the form

$$B_1 \sqcap \dots \sqcap B_n \rightarrow B' \quad \text{and} \quad \exists r.B$$

with B_1, \dots, B_n, B , and B' concept names, r a role name, and $n \geq 0$. For the case $n = 0$, we simply write the queue entry $B_1 \sqcap \dots \sqcap B_n \rightarrow B'$ as B' . Intuitively,

- an entry $B_1 \sqcap \dots \sqcap B_n \rightarrow B'$ in $\text{queue}(A)$ means that B' has to be added to $S(A)$ if $S(A)$ already contains B_1, \dots, B_n , and
- $\exists r.B \in \text{queue}(A)$ means that (A, B) has to be added to $R(r)$.

The fact that such an addition triggers other rules will be taken into account by appropriately extending the queues when the addition is performed.

To facilitate describing the manipulation of the queues, we view the (normalized) input ontology \mathcal{O} as a mapping $\widehat{\mathcal{O}}$ from concepts to sets of queue entries as follows: for each concept name $A \in \text{CN}_{\mathcal{O}}^{\top}$, $\widehat{\mathcal{O}}(A)$ is the minimal set of queue entries such that

- if $A_1 \sqcap \dots \sqcap A_n \sqsubseteq B \in \mathcal{O}$ and $A = A_i$, then

$$A_1 \sqcap \dots \sqcap A_{i-1} \sqcap A_{i+1} \sqcap \dots \sqcap A_n \rightarrow B \in \widehat{\mathcal{O}}(A);$$

```

procedure process( $A, X$ )
begin
  if  $X = B_1, \dots, B_n \rightarrow B$  and  $B \notin S(A)$  then
    if  $\{B_1, \dots, B_n\} \subseteq S(A)$  then
       $S(A) := S(A) \cup \{B\}$ ;
       $\text{queue}(A) := \text{queue}(A) \cup \widehat{\mathcal{O}}(B)$ ;
      for all concept names  $A'$  and role names  $r$ 
        with  $(A', A) \in R(r)$  do
           $\text{queue}(A') := \text{queue}(A') \cup \widehat{\mathcal{O}}(\exists r.B)$ ;
    if  $X = \exists r.B$  and  $(A, B) \notin R(r)$  then
      process-new-edge( $A, r, B$ )
end;

procedure process-new-edge( $A, r, B$ )
begin
  for all role names  $s$  with  $r \sqsubseteq_{\mathcal{O}}^* s$  do
     $R(s) := R(s) \cup \{(A, B)\}$ ;
     $\text{queue}(A) := \text{queue}(A) \cup \bigcup_{\{B' | B' \in S(B)\}} \widehat{\mathcal{O}}(\exists s.B')$ ;
    for all concept names  $A'$  and role names  $t, u$  with
       $t \circ s \sqsubseteq u \in \mathcal{O}$  and  $(A', A) \in R(t)$  and  $(A', B) \notin R(u)$  do
      process-new-edge( $A', u, B$ );
    for all concept names  $B'$  and role names  $t, u$  with
       $s \circ t \sqsubseteq u \in \mathcal{O}$  and  $(B, B') \in R(t)$  and  $(A, B') \notin R(u)$  do
      process-new-edge( $A, u, B'$ );
end;

```

Figure 3: Processing the queue entries

- if $A \sqsubseteq \exists r.B \in \mathcal{O}$, then $\exists r.B \in \widehat{\mathcal{O}}(A)$.

Likewise, for each concept $\exists r.A$, $\widehat{\mathcal{O}}(\exists r.A)$ is the minimal set of queue entries such that, if $\exists r.A \sqsubseteq B \in \mathcal{O}$, then $B \in \widehat{\mathcal{O}}(\exists r.A)$.

In the modified algorithm, the queues are used as follows: since the sets $S(A)$ are initialized with $\{A, \top\}$, we initialize $\text{queue}(A)$ with $\widehat{\mathcal{O}}(A) \cup \widehat{\mathcal{O}}(\top)$, i.e., we add to the queues the *immediate* consequences of being an instance of A and \top . Then, we repeatedly fetch (and thereby remove) entries from the queues and process them using the procedure **process** displayed in Figure 3. To be more precise, **process**(A, X) is called when the queue of A was non-empty and we fetched the queue entry X from $\text{queue}(A)$ to be treated next. Observe that the first if-clause of the procedure **process** implements **R1** and (part of) **R3**, and the second if-clause implements **R2**, (the rest of) **R3**, as well as **R4** and **R5**. The procedure **process-new-edge**(A, r, B) is called by **process** to handle the effects of adding a new pair (A, B) to $R(r)$. The notation $\sqsubseteq_{\mathcal{O}}^*$ used in its top-most **for**-loop stands for the reflexive-transitive closure of the role hierarchy axioms in \mathcal{O} . Queue processing is continued until all queues are empty. Observe that the refined algorithm need not perform *any* search to

check which completion rules are applicable.

This is the major and most novel optimization implemented in the CEL reasoner. Other optimizations that have helped enhance the performance are listed below:

- Reuse of new concept names. A complex concept term may occur twice or more in an ontology; a unique name is introduced per concept term instead of per occurrence.
- Encoding of user concept and role names. For faster internal processing, potentially long names are encoded into fixed integers, which are decoded back to the original names only when the user demands output.
- Optimized computation of subsumption hierarchy from the completed implication sets. In short, we consider the implication sets as complete information about told subsumption and adopt a simplified version of the known classification method with told information [BFH⁺94].

4.3 How to use

The CEL system is available as a binary executable which can run on most Linux platforms. The latest version is CEL v0.94 which includes all features illustrated in this system description. The distribution bundle can be obtained from:

<http://lat.inf.tu-dresden.de/systems/cel/>

The package consists of the CEL executable, the user manual [Sun05a], and some toy \mathcal{EL}^+ ontologies. After extracting the bundle, the executable `cel` under `./bin` can be started up without need for installation. However, the following system requirements are assumed:

- Linux operating system;⁴
- Physical memory at least 128MB;⁵
- At least 8MB of available hard-disk space.⁶

Essentially, there are two modes of operations: stand-alone reasoner and backend server. Backend reasoning mode has an advantage over the other mode in that the users are insulated from technical hassles and potentially incomprehensible output messages. Moreover, the CEL reasoner may be installed on a high-end dedicated computing server, while the actual user computer may run an application that exploits services from CEL. On the other hand, stand-alone mode of operation avoids unnecessary overheads, e.g. communication and parsing, and as a result, is much more efficient and scalable.

⁴It has been tested successfully on RedHat, Debian, and SuSE.

⁵Considerably more memory may be needed for larger ontologies.

⁶Of course, much more disk space is required to archive classification results (either subsumer sets, parent-child relationships or subsumption hierarchy). This could or could not be speculated from the size of the input ontology.

Ontology axioms	DL Syntax	CEL Syntax
primitive concept definition	$A \sqsubseteq D$	(define-primitive-concept $A D$)
concept definition	$A \equiv D$	(define-concept $A D$)
general concept inclusion	$C \sqsubseteq D$	(implies $C D$)
concept equivalence axiom	$C \equiv D$	(equivalent $C D$)
concept disjointness axiom	$C \sqcap D \sqsubseteq \perp$	(disjoint $C D$)
role domain axiom	$\text{dom}(r) \sqsubseteq C$	(define-primitive-role r :domain C)
role range axiom	$\text{ran}(r) \sqsubseteq C$	(define-primitive-role r :range C)
role hierarchy axiom	$r \sqsubseteq s$	(define-primitive-role r :parent s)
transitive role axiom	$r \circ r \sqsubseteq r$	(define-primitive-role r :transitive t)
right-identity axiom	$r \circ s \sqsubseteq r$	(define-primitive-role r :right-identity s)
left-identity axiom	$s \circ r \sqsubseteq r$	(define-primitive-role r :left-identity s)
complex role inclusion	$r_1 \circ r_2 \sqsubseteq s$	(role-inclusion (compose $r_1 r_2$) s)

Table 1: The CEL Syntax for \mathcal{EL}^+ ontology axioms.

CEL as a stand-alone reasoner. In order to use CEL to classify an ontology, the user must already have the ontology formulated in \mathcal{EL}^+ in a small extension of the KRSS syntax [PSS93], henceforth called CEL syntax. With this LISP-like syntax, it is easy to port existing ontologies that have been used with well-known DL reasoners like RACERPRO and FaCT. For building up ontologies, the expressive means shown in Table 1 can be used, where conventionally A, B denotes a named concept, C, D concept descriptions, and r, s named roles. Though only **implies** and **role-inclusion** axioms can sufficiently model any \mathcal{EL}^+ ontology, it is often very useful and also makes the ontology more comprehensible to provide auxiliary axioms. An \mathcal{EL}^+ ontology is effectively a text file containing axioms of the forms shown in the right column of Table 1.

As an example, the toy ontology in Figure 2 formulated in the CEL syntax can be found in the distribution bundle under `./tbox/med.tbox`. The user can either load this ontology into the system by calling `(load-ontology "med.tbox")` or enter interactively at the prompt each axiom from the ontology. The preprocess is carried out while the ontology is being loaded, and once this is finished, `(classify-ontology)` can be invoked to classify all concept names occurring in the ontology (eager subsumption approach). Subsumption query between two concept names can be queried using `(subsumes? B A)`. If this is called after classification, it simply looks up in the computed subsumption hierarchy. Otherwise, it runs a single subsumption test and answers without needing to classify the whole ontology first (lazy subsumption approach). After having classified the whole ontology, CEL allows the user to output the classification results in different formats: `(output-supers)` to output the subsumer sets for all concept names occurring in the ontology; `(output-taxonomy)` to output the Hasse diagram of the subsumption hierarchy, i.e., parent-child relationships; and `(output-hierarchy)` to output the hierarchy as a graphical indented tree. As an example, Figure 4 depicts screen shots of the results of `(output-hierarchy)` and `(output-taxonomy)` after classifying the ontology `med.tbox`.

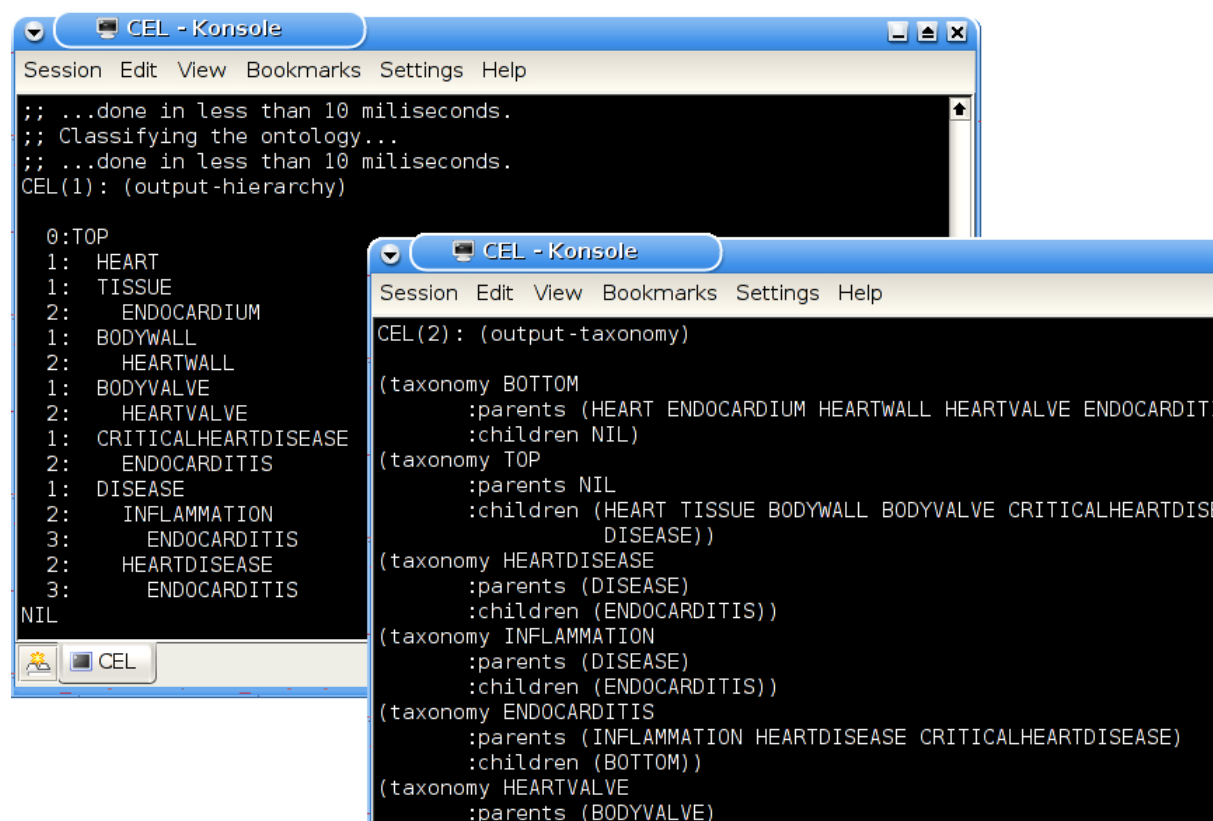


Figure 4: CEL with its innate interactive interface

Through its command-line options, CEL can also work as a stand-alone reasoner without interaction from users. For instance, the command line:

```
$cel -l file -c -outputHierarchy -q
```

can be entered to load and classify an ontology from *file*, and then output the hierarchy. For a more detailed description of the CEL interface, we refer to the CEL user manual [Sun05a].

CEL as a backend reasoner. Alternatively, the user can also exploit CEL reasoning capabilities through the DIG interface⁷ and a graphical ontology editor. To do this, CEL has to be started as a DIG reasoning server by the following command line:

```
$cel -digServer [port]
```

where *port* is defaulted to 8080 but can be overridden.

Once started in this mode, an ontology editor can connect to CEL and exploit its reasoning services either locally or remotely via the Internet. The upper floating dialog in Figure 5, “Reasoner Inspector,” displays the expressive means that can be handled by CEL

⁷The DIG (DL Implementation Group) interface is an XML-based standard that defines an interfacing language for seamless communication between a DL service provider (DIG server) and a DL application (DIG client). See <http://dl.kr.org/dig/>

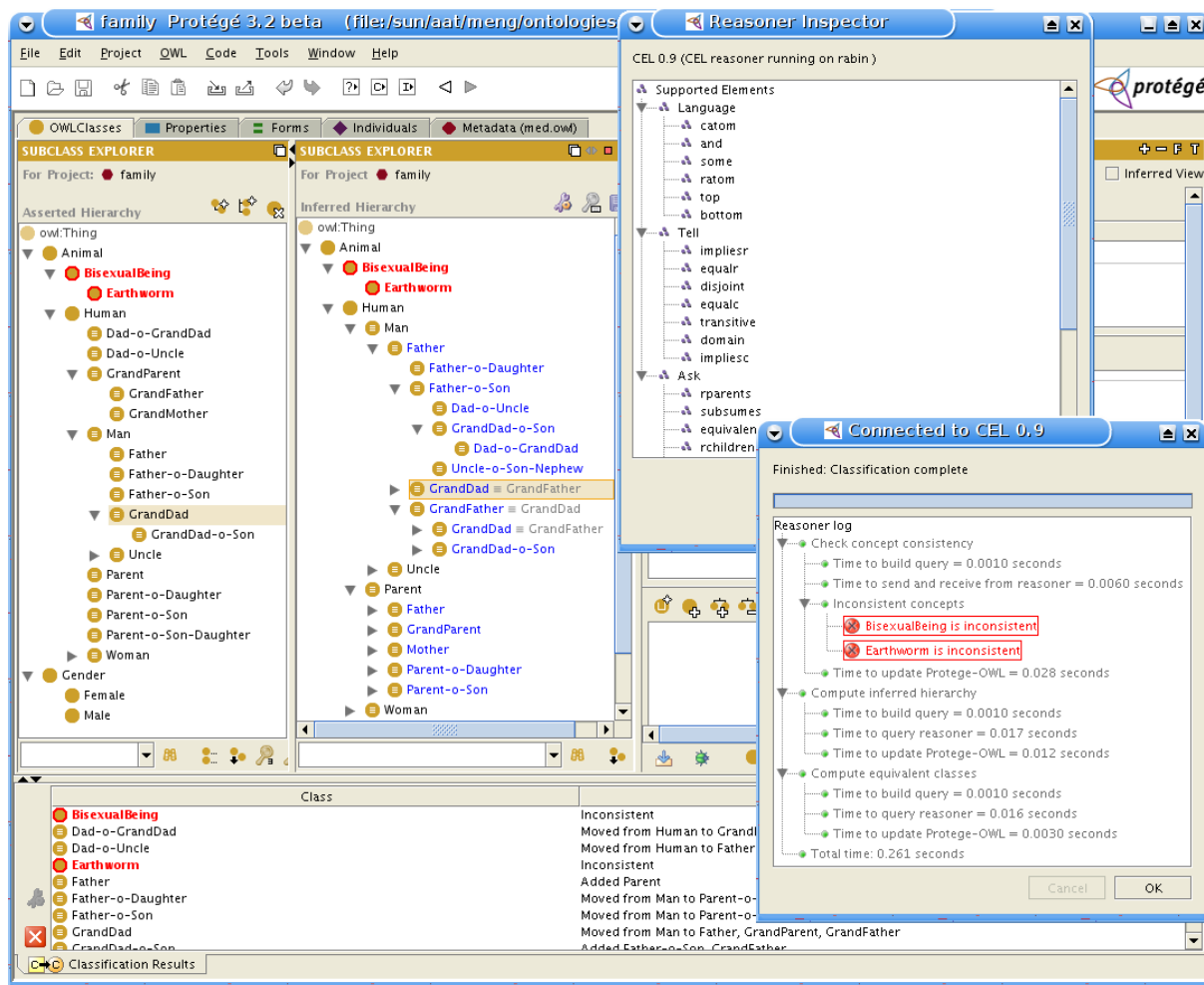


Figure 5: CEL as a DIG reasoner supporting the Protégé OWL editor

in terms of DIG language. The DIG interface for CEL has been tested successfully with Protégé OWL editor.⁸ The main window in Figure 5 illustrates the asserted subsumption hierarchy (input) and the inferred subsumption hierarchy (output) within the editor, whereas the small floating dialog, “Connected to CEL 0.9,” displays an interaction log between the DIG client and the DIG server.

⁸See <http://protege.stanford.edu/plugins/owl/>

5 Swoop

5.1 Introduction

Swoop is built primarily as a *Web Ontology* Browser and Editor, i.e., it is tailored specifically for OWL ontologies. Thus, it takes the standard Web browser as the UI paradigm, believing that URIs are central to the understanding and construction of OWL Ontologies. The familiar look and feel of a browser emphasized by the address bar and history buttons, navigation side bar, bookmarks, hypertextual navigation etc are all supported for web ontologies, corresponding with the mental model people have of URI-based web tools based on their current Web browsers.

All design decisions are in keeping with the OWL nature and specifications. Thus, multiple ontologies are supported easily, various OWL presentation syntax are used to render ontologies, open-world semantics are assumed while editing and OWL reasoners can be integrated for consistency checking. A key point is that the hypermedia basis of the UI is exposed in virtually every aspect of ontology engineering — easy navigation of OWL entities, comparing and editing related entities, search and cross referencing, multimedia support for annotation, etc. — thus allowing ontology developers to think of OWL as just another Web format, and thereby take advantage of its Web-based features.

A diverse array of ontology related tasks can be performed in Swoop, namely⁹:

- Authoring concept descriptions and axioms,
- Structuring the ontology, and
- Error management.

Swoop is accessible to both, novice users interested in casual ontology building and use, and expert users interested in robust ontology modeling and analysis. The development of Swoop started at the University of Maryland in 2004. Within TONES, the error management capabilities of the editor have been improved and new features concerning ontology modularization and reuse have been implemented. Finally, Swoop has been extended to support OWL 1.1.

Swoop can be downloaded from at the following address: <http://www.mindswap.org/2004/SWOOP>

5.2 Swoop Features

In this section, we describe the features of Swoop that are in keeping with its design rationale and goals mentioned earlier.

5.2.1 Ontologies based on the Web Architecture

The idea behind Web ontology development is different from traditional and more controlled ontology engineering approaches which rely on high investment, relatively large, heavily engineered, mostly monolithic ontologies. For OWL ontologies, which are based on

⁹See Deliverable D13 for a description of these tasks.

the Web architecture (characterized as being open, distributed and scalable), the emphasis is more on utilizing this *freeform* nature of the Web to develop and share (preferably smaller) ontology models in a relatively ad hoc manner, allowing ontological data to be reused easily, either by linking models (using the numerous mapping properties available in OWL) or merging them (using the `owl:imports` command). Thus, it becomes essential for a Web ontology development tool to *scale* to multiple ontologies easily, and to allow tasks such as creation, browsing, editing, search, reuse, linking, merge/split of OWL ontology models in the context of multiple ontologies.

In order to attain this key requirement, Swoop ensures that users are free to load multiple OWL ontologies in any manner they prefer. The easiest and most direct way to load an OWL Ontology is by entering its physical URL (Web or local file address) in the address bar. This action not only pulls in the requested ontology, but also loads any imported ontologies (defined using `owl:imports`) into Swoop automatically. The **bookmarks** feature can be used to store, categorize and reload ontologies directly (as is the case in standard web browsers). Finally, depending on user preference, an ontology can also be brought into Swoop rather seamlessly during browsing/editing, e.g., attempting to view or refer to an externally referenced entity while in a particular ontology can load the external ontology automatically.

There are certain characteristics of OWL ontologies which are presented to the user when a new ontology is brought into Swoop. The Ontology Renderer plugins in Swoop accomplish this, and display statistics such as (see **Fig. 6**):

- the logical constructs used in the ontology model which determine the OWL species level the ontology belongs to, i.e., OWL Lite, DL or Full
- the Description Logic (DL) expressivity of the ontology - a key factor in determining decidability of reasoning
- number of classes, properties, individuals etc. (we intend to extend the granularity to axioms, e.g., no. of disjoint axioms, no. of nominals used etc.)
- annotations on the ontology object itself (including `owl:imports`)

5.2.2 Editing Web Ontologies

Consider a scenario in which a user is building an ontology for describing the administrative hierarchy (with concepts such as **Department**, **Faculty**, **Staff**, **Student** etc) of a university. This user can make use of existing concepts in well-known upper-level ontologies such as FOAF or Cyc (for generic concepts such as **Person**), or in similar ontologies created for other universities. Another user interested in building a finer-grained ontology than the one above, say for describing his/her research group and can now use the university ontology to refer to or define certain concepts. In this manner, the open development cycle of *create-link-share* web ontologies ensures that a large amount of interrelated semantic content is available in ontologies.

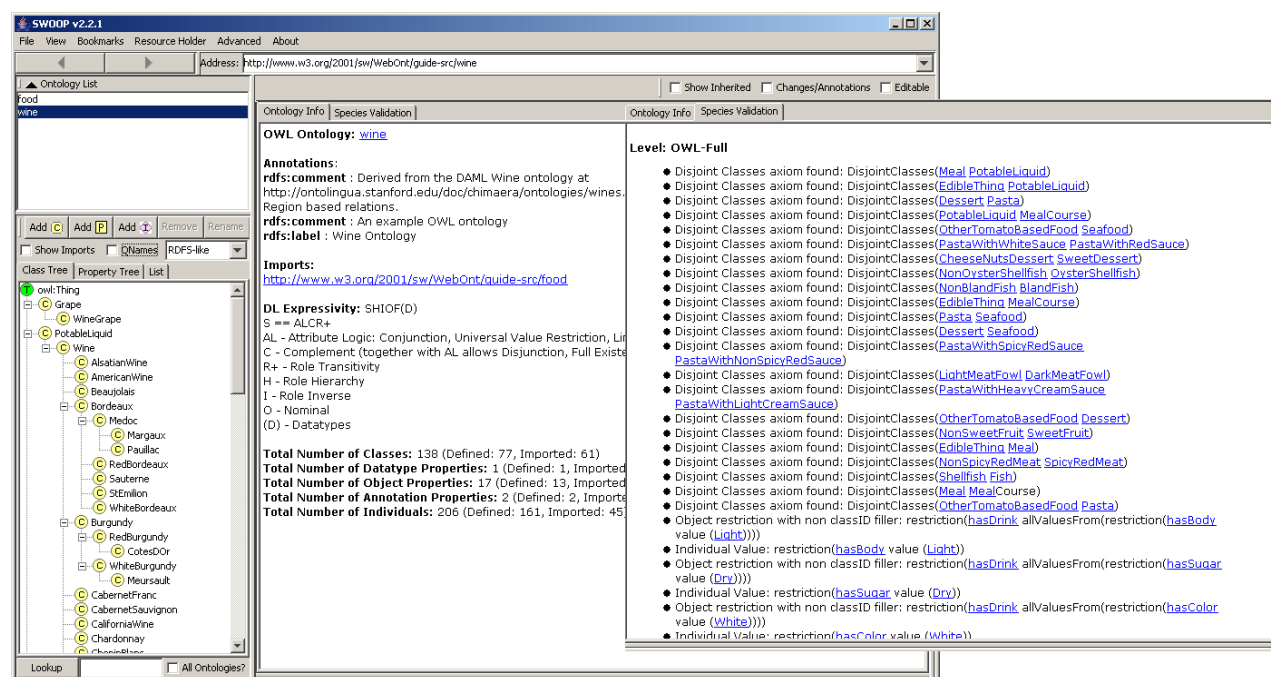


Figure 6: **Swoop Ontology Renderers**: display statistical information about the ontology, annotations, DL expressivity and OWL species level.

In keeping with the above scenario, Swoop allows users to freely link (map between) entities in different ontologies using a single common interface, which lists each ontology loaded in Swoop along with its corresponding entity list (see **Fig: 7**)¹⁰

However, there are additional caveats to be considered while editing in a multiple ontology setting as described above. For starters, it is essential to provide a search feature to help users find related ontological information. Having found such information, it then becomes critical to compare and analyze this information in order to determine which parts, if any, are useful (verifying relevance, accuracy etc). Finally, the user needs a flexible reuse scheme that supports either borrowing the entire external ontology model if desired, or a subset of it which is relevant, allowing suitable modifications if any. We deal with each of these three caveats in detail as reflected in Swoop.

Search in Swoop essentially performs a lookup for entities (classes/ properties/ individuals) across single or multiple ontologies, among those that have been loaded. The results are obtained as a set of hyperlinks (in keeping with the hypermedia-based UI) allowing the user to browse the search results easily.

During an extensive search/browsing process, the user may need to set aside and revisit interesting search results. In Swoop we have a provision to store and compare OWL entities via a **Resource Holder** panel. Items can be added to this panel at any time and they remain static there until the user decides to remove or replace them

¹⁰It is important to consider the scenario in which a user edits an external ontology present at a URL under the control of a third party. In this case, a local version of the ontology is maintained separately and its physical location is used for reference in an `owl:import` axiom that specifies importing the external ontology.

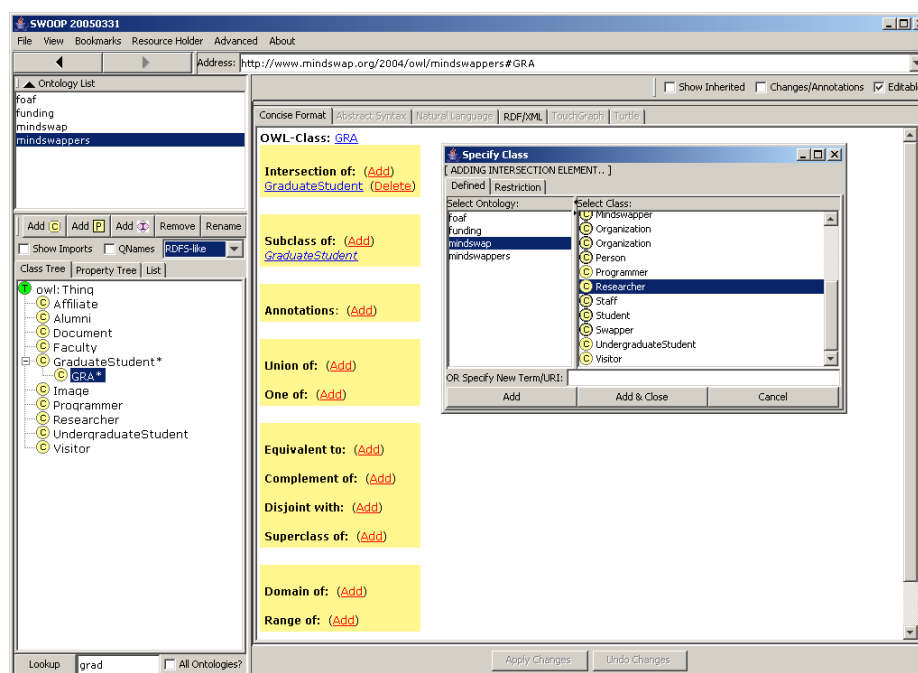


Figure 7: **Editing in Swoop:** Clicking the ‘Add’ hyperlink next to an assertion heading (e.g., **Intersection of**) pops up a window to specify corresponding new information (e.g., the new intersection class). In this case, the user is specifying a class **Researcher** from an external ontology as an intersection element

at a later stage. This common placeholder acts as an excellent platform for performing interesting engineering tasks such as comparing differences in definitions of a set of entities; determining semantic mappings between a specific pair of entities or simply storing entities for reusing in another ontology.

5.2.3 Adhering to OWL Specifications: Presentation and Reasoning

Currently, various presentation syntax exist for rendering OWL ontologies such as RDF/XML and OWL Abstract Syntax. It is important to support these different syntax while designing an open, Semantic Web ontology engineering environment. One reason for this is that people tend to have strong biases toward different notations and simply prefer to work in one or another. A second is that some other tool might only consume one particular syntax (with the RDF/XML syntax being the most typical), but that syntax might not be an easy or natural one for a particular user. A third is that it is important to support the “view source” effect, allowing cut and paste reuse into different tools including text editors, markup tools, or other semantic web tools. For these reasons, Swoop has default plugins for all three presentation syntax mentioned above. Users are free to browse and edit ontological data, either at the level of a single entity (inline) or at the level of the entire ontology as a whole, in any syntax as desired, switching between syntax on a single click.

In addition to the default OWL presentation syntax, we are working on three additional renderers to help users visualize and understand OWL ontologies better. These

include a **Concise Format** entity renderer, where the idea is to generate a “Web document” that displays all information related to a particular OWL entity concisely in a single pane; a **Natural Language** entity renderer that provides concise, accurate NL paraphrases for OWL Concepts based on a variety of NLP techniques; and an **OWL Graph visualization** renderer based on TouchGraph that displays concise conceptual graphs of the ontology model. Each of these renderers provide a different view of the model, allowing users to understand logical definitions and relationships better.

5.2.4 Reasoning in OWL

Having covered the presentation aspects of OWL ontologies, we now focus on the *reasoning* support in Swoop. Note that OWL-DL is primarily based on description logic, with open-world semantics and a non unique name assumption (UNA). Swoop strictly maintains the latter two aspects during editing, e.g., it does not try to ‘interfere’ with creating the KB (i.e., prevent the creation of inconsistencies) by making any additional alterations or assumptions, and accurately reflects the users’ actions based on open world semantics. As for the DL reasoning, Swoop allows for special-purpose reasoner plugins that provide standard reasoner services such as satisfiability (of a single class as well as consistency of the ontology), subsumption (between classes and between properties), and realization (types of an instance). Additionally, reasoners can support the optional *explanations* feature, which is used for sophisticated ontology debugging as explained later.

Swoop contains two additional reasoners (besides the basic *Reasoner* that simply uses the asserted structure of the ontology): *RDFS-like* and *Pellet* (<http://pellet.owldl.com/>). While the former is a lightweight reasoner based on RDFS semantics, the latter, Pellet, is a powerful description logic tableaux reasoner. Pellet has a number of advantages: It natively supports OWL, including a repairable subset of OWL Full; it has extensive support for XML Schema datatypes; it has ABox (a.k.a., instance) support; it covers the broadest range of OWL DL of any reasoner that we know, including both *SHIN*(\mathcal{D}), *SHON*(\mathcal{D}), *SHIO*(\mathcal{D}), and various subsets of their union, *SHOIN*(\mathcal{D}) (a.k.a., OWL DL); it is open source and in active, public development. The last is very important for certain debugging strategies which require access to the internals of the reasoner as noted later.

The above reasoners provide a tradeoff between speed and quality of inference results, e.g., the *RDFS-like* reasoner, while much faster than Pellet in execution, is unsound (results maybe inaccurate if the ontology is inconsistent) and incomplete (does not list all possible inferences). Yet, in most cases, it provides interesting and useful results for ontology authors, and moreover, the reasoners can be used in conjunction to analyze the ontology quickly while editing it.

5.2.5 Ontology Debugging and Repair

DL reasoners can be used to detect inconsistencies in definitions of concepts (a.k.a. unsatisfiable concepts). However, typically reasoners only report that a class is unsatisfiable, not *why*. Moreover, they do not report on inter-dependencies (if any) of the unsatisfiable classes, i.e., if a class directly depends on another for its unsatisfiability (e.g., by an existential property restriction on an unsatisfiable class). We argue that both forms of

explanation are essential for the purpose of debugging ontologies; while the former can be used to understand and rectify problematic axioms / class expressions, the latter can help prune out dependency bugs and let the modeler focus on the root (source) of the problem alone.

We distinguish two families of reasoner-based techniques for supporting diagnosis of the form described above: glass box and black box techniques. In glass box techniques, information from the *internals* of the reasoner is extracted and presented to the user (typically used to pinpoint the type of clash/contradiction and axioms leading to the clash). In black box techniques, the reasoner is used as an oracle for a certain set of questions e.g., the standard description logic inferences (subsumption, satisfiability, etc.) and the asserted structure of the ontology is used to help isolate the source of the problems (can be used to find dependencies between unsatisfiable classes). Swoop currently also provides support for repairing unsatisfiable concepts (see D13 for details).

5.3 How to Use

In order to download the software, access the Swoop homepage at <http://www.mindswap.org/2004/SWOOP>. After unzipping the downloaded file (SWOOP-xxxx.zip), execute `runme.bat` (/ `runme.sh`) present inside the "SWOOP-xxxx" directory to start the application on a Windows (/ Mac or Unix) machine. For loading large ontologies such as NCI, you need to allocate more memory for Swoop - use the `runme-HIGH` file in this case.

The application requires Java 1.4 installed on your machine. You can download the latest version of Java from <http://java.sun.com/j2se/1.4/download.html>

The SWOOP application includes/uses the following API's:

- WonderWeb OWL API (<http://sourceforge.net/projects/owlapi>) MODIFIED SOURCE (see changelog at the end of source file).
- XNGR API (<http://xngr.org/>)
- Jakarta Slide WebDAV API (<http://jakarta.apache.org/slide/>)
- QTag API (<http://www.english.bham.ac.uk/staff/omason/software/qtag.html>).
- Hexidec Ekit API (<http://www.hexidec.com/ekit.php>) MODIFIED SOURCE (see changelog at the end of source file) which are located in the `/lib` sub-directory under SWOOP. Additional jars in the `lib` directory (if present) are plugin dependencies.

SWOOP employs a plugin based system for easy extension. Sample plugins can be downloaded from <http://www.mindswap.org/2004/SWOOP/plugins>.

6 RacerPorter

6.1 Introduction

RACERPORTER is a text-based ontology editor and the default GUI client of the RACERPRO description logic system (DLS). The metaphorical name RACERPORTER was chosen to stress that a “user friendly entrance” shall be provided to an otherwise “faceless” DL-server, like a hotel porter. Although quite a number of ontology browsing and inspection tools (called OBIT in the following) as well as authoring tools exist and numerous papers have been written about them [KPS⁺05, LN05, LN06, KMR04], RACERPORTER represents a different approach. We present the design principles behind RACERPORTER as well as the tool.

As already mentioned before (e.g., in [LBF⁺06]), ontology editors are currently the main software tool for ontology design tasks. They provide for functionality such as browsing and editing single ontology elements and the whole ontology structure, performing communication with background reasoners, visualization of reasoners’ feedback and so on.

When developing RACERPORTER, the aim was not only to support this basic functionality but also to enhance usability and to solve certain “scalability problems”. Users “unscrupulously” load rather large OWL files into the reasoner and expect their taxonomies to be visualized with the ontology design tools such as RACERPORTER. We reacted to the complaints of RACERPORTER users by enhancing the performance and usability of previous versions of RACERPORTER on large KBs.

RACERPORTER exclusively uses the KRSS port of RACERPRO, although support for OWL is included as well. Compared with DIG, KRSS has the advantage that it can also be used as a *shell language* (DIG was designed under a different perspective). The XML messages standardized by DIG are not on the correct level of abstraction for a shell language (even if a non-XML serialization of DIG messages were used).

In a nutshell, RACERPORTER has been designed to meet the following design characteristics:

1. RACERPORTER offers a KRSS shell for interactive communication with RACERPRO. Already RICE (visit <http://www.ronaldcornet.nl/rice/>) offered a shell.
2. Unlike tools such as OntoTrack [LN05, LN06], Swoop [KPS⁺05] and GrOWL [SK06], RACERPORTER is not designed to be a graphical OWL authoring tool. However, we believe that for expert users also text-based editors are useful and these text editing facilities have to be tightly integrated with graphical visualization tools. In RACERPORTER, we added a text editor with Emacs-styled buffer evaluation mechanisms which in combination with a shell allows for rapid and interactive authoring of KRSS KBs.
3. Obviously, ontology visualization is important as well. Ontologies have different aspects, i.e., intensional and extensional ones. One can expect that an OBIT is able to visualize taxonomies, role hierarchies as well as ABoxes as graphs and/or trees (of certain kind, using certain graph layout algorithms). An OBIT should thus provides appropriate visualization facilities.

4. Given the fact that OWL KBs tend to become bigger and bigger, appropriate navigation, browsing and focusing mechanisms must be provided, since otherwise the user gets “lost in ontology space”. An OBIT must thus provide appropriate (syntactic and semantic) mechanisms.
5. OBITs (as well as the corresponding DLSs, of course) should be able to process very large ontologies (scalability aspect).
6. Given that both shell-, gadget- as well as graph-based interactions are offered, the question arises: How to link these interactions, and the results produced by them? An OBIT must provide appropriate solutions.
7. An OBIT should be designed to work in a non-blocking way (asynchronously). This is especially valuable if large ontologies are processed, and processing takes some time.
8. It is desirable if an OBIT can maintain different connections simultaneously to different DLS servers. While one server is busy, the user can change the active connection and continue work with another server.
9. An OBIT should avoid opaqueness. Especially if modes are used (and the interface is stateful), then it is necessary to appropriately visualize these modi.
10. Functionality for starting, stopping and controlling DL servers is desirable. Since each DLS has its proprietary functions and peculiarities, it becomes clear that at least part of the OBIT functionality must be tailored for the target DLS.

6.2 Towards User-Friendly and Scalable OBITs

A KRSS or OWL ontology represented in a DLS has many different aspects: the taxonomy represents the subsumption relationships between concept names or OWL classes, the role hierarchy represents the subsumption relationships between roles or OWL properties, and the ABox represents information about the individuals and their interrelationships (the extensional knowledge). Additional aspects may be present, e.g. queries and rules. Thus, we can make a “shopping list” of “things” which must be accessed, managed and visualized with a DLS OBIT: different DL servers and their connection profiles¹¹, TBoxes, ABoxes, concepts, roles, individuals, queries and rules, ABox assertions, etc.

In order to avoid an overloaded GUI – which would try to represent these different aspects and aspect-specific functionality in a single window pane – in a similar way as other graphical ontology tools, we favor tabbed interfaces in order to achieve a clean separation of different aspects. Different tabs thus present different aspects of the ontology together with aspect-specific commands. The term “different perspectives” also describes the approach quite well.

Whereas many operations are directly performed on the displayed representations of the objects on the RACERPRO servers by means of mouse gestures (direct manipulation),

¹¹The connection and server settings can be managed using the so-called connection profiles which are familiar from networking tools such as SFTP browsers.

we also favor push buttons to invoke commands. In many cases, push buttons will directly invoke KRSS commands, e.g., send an `abox-consistent?` to the connected DL server. Push buttons also have the neat effect to inform the user directly about commands which are reasonable to apply or which can be applied at all in a given situation, simply by being visible, so there is no need to search for a command in a pull-down menu, which distracts focus. However, it should also be noted that the buttons occupy screen space, and using buttons and menus should be kept in balance.

In many cases, some input arguments must be provided to KRSS commands. Input arguments are provided directly by the user if direct manipulation is employed for the interaction, but with simple push buttons this is not directly possible. Either the user must be prompted for arguments, or a notion of “current objects” must be employed. These current objects may have been (implicitly) selected by the user before and are from then on automatically supplied as input arguments to KRSS functions. This results in a stateful GUI. Sometimes, stateful GUIs are considered harmful. However, we will see that states are unavoidable if non-trivial ontology-inspection tasks shall be performed. Additionally, since also a DLS has a state, this state should be adequately reflected by the GUI as well (which automatically makes it stateful). In order to avoid opaqueness it is very important that the current state is appropriately visualized, e.g., in a status display which is visible at any time. According to the shopping list mentioned before, we must thus have a notion of a current DLS server, a current TBox and ABox, current concept, individual and role, current query, etc. These current objects partially constitute the current state of the OBIT.

The different tabs of the OBIT visualize often different objects. For example, one tab shows the individuals in the current ABox (the individuals tab), and another tab shows the concepts in the current TBox (the concepts tab). The information displayed in a certain tab thus depends on the current state. Additionally, the current concept (or the current individual) will be highlighted in the concepts tab (resp. the individuals tab), so it can be recognized easily. Two different tabs can also present the same objects, but use different visualizations. For example, the taxonomy tab also presents the concepts in the current TBox, but displays them as nodes in a graph whose edges represent subsumption relationships. Since all tabs display information according to the current state, the shown information is interrelated.

For certain ontology inspection tasks, it is further necessary to relate the information displayed on different tabs. One must establish a kind of information flow between different tabs. Let us illustrate this need for an input/output flow of information with an example.

As described, the individuals tab presents the list of individuals in the current ABox. If an individual is selected from that list, it automatically becomes the current individual. In order to explore of which concepts this individual is a direct instance, it is possible to push the “Direct Types” button which sends the `direct-types` KRSS command to the DLS, using the current ABox and current individual as arguments. In many cases, further operations shall be applied to the result concepts just returned, e.g., in order to explore which other instances of these concepts are presented in the KB. Thus, the concepts tab should provide a functionality which allows to refer to and highlight the just returned concepts, so that subsequent operations can be easily applied on them.

In order to establish this kind of information flow, we augment the notion of the current state by also including sets of selected objects in the state. Thus, the concepts returned by the `direct-types` command can become selected concepts. Selected concepts are shown as highlighted, selected items in the tabs which present concepts. Moreover, there are also selected individuals and selected roles. Objects can either be selected manually by means of mouse gestures, or automatically by means of KRSS commands, no matter how they are invoked. All what matters is the notion of selected objects. The set of selected objects is also called the clipboard. The current objects are seen as specific selected objects. Furthermore, all this state information is session-specific, given the fact that the OBIT should be able to maintain several connections and thus associated sessions simultaneously.

As said earlier, a shell tab is provided for interactive textual communication with the DLS. We claim that only shell-based interactions can offer the required flexibility and expressivity needed for advanced ontology inspection tasks. The shell must be incorporated into the above mentioned information flow as well. For example, if the `direct-types` command is entered into the shell, then it must be possible to refer to the current ABox as well as to the current individual which has been selected with the mouse in the individuals tab before (without having to type its name or having to use “copy & paste”). Furthermore, after the command was executed, it must be possible to select the returned concepts which are now shown in the shell as command output.

Focus control and navigation are two other important issues. It is well-known that the notion of current and selected objects can be used to control the focus. For example, the current concept can provide the root node in the taxonomy graph display. Only the descendants of the current concept will be shown. To browse larger taxonomies, a “depth limit” cutoff on the paths to display can be specified, and an interactive “drill down”-like browsing style can be realized. The node gesture “select” (e.g., a left mouse click) automatically changes the current concept and thus the graph root. If the graph is redrawn immediately, this allows to drill down a large taxonomy interactively and dynamically. However, this automatic graph recomputation changes the focus.

In principle, changing the focus automatically can be very distracting. In Web browsers, the navigation buttons (back and forth) are thus of utmost importance; they allow to reestablish the previous focus effortlessly. Thus, a focus or history navigator should be present in an OBIT, as also found in Swoop or GrOWL [KPS⁺05, SK06]. However, many users are unhappy with hyperlink-like focus-destroying operations. In Web browser, tabbed browsing has been invented to address this problem. Thus, we think that the user should be able to determine when and how the focus is changed once a gadget is selected.

Sometimes, it is also desirable to focus on more than one object, e.g., for ABox graphs. We can simply use the selected objects for that as well. In case of the ABox graph, each selected individual can specify a graph root, and unraveling (as understood in Modal Logics) is used to establish a local perspective from that individual’s point of view (so there is one graph per selected individual). This resolves many visual cluttering problems. The clipboard is thus not only a structure that enables flow of information, but can also be used to control the focus. This also implies that the focus control is now highly flexible: Since the clipboard can be filled from results of KRSS commands, even a semantic focus

control is possible. For example, an ad-hoc nRQL query can be typed into the shell, and, with the push of a button, one can focus on the returned ABox individuals in the ABox graph tab. In our terminology, this kind of focusing by invoking inference services (such as, e.g., `direct-types`) is called semantic focusing. However, one also often wants to focus on individuals simply by their names. Thus, a kind of search field is needed. Objects that contain the search string get selected automatically. This enables a so-called syntactic focusing. We have found that many available tools don't offer adequate mechanisms to achieve this kind of information flow and focus control.

Summing up, we conclude that the current state must be a vector `<current objects,selected objects,active tab,display options>`. Each time a state changing operation is performed by the user (e.g., the current objects or the clipboard is changed), a so-called history entry is automatically created, which is just a copy of the current state vector. History entries are inserted at the end of a queue, the so-called navigation history. A history navigator offers the standard navigation buttons. The OBIT always reflects the current state, no matter whether this is the latest one or a historic state from the history. A history entry only preserves the state information of the GUI, but not the content of the DLS at that time. Thus, a well-known problem arises here: If a historic state is reactivated, then it may no longer be possible to actually refer to the objects referenced by that state, since they may have been already deleted from the DLS. This problem is well-known to WWW users which keep a browsing history. There is no practical solution to this problem (one cannot preserve “copies” of DLS server states in history entries).

We believe that OBITs should allow for asynchronous usage. While a time-consuming command is processed by the DLS, the GUI shouldn't block; instead, the result should be delivered and displayed asynchronously once available. Although the busy DLS will not accept further commands until the current request had been fulfilled (nowadays, there are no true multi-user DLS), in the meantime the OBIT should a) display status information in order to inform the user what the DLS is currently doing (future versions of RACERPRO and RACERPORTER will also support progress bars), and b), if possible, allow the user to do other things, e.g., continue editing a KRSS KB, or connect to and work with a different DLS.

6.3 RacerPorter – How to Use

RACERPORTER was designed according to the design principles just explained. Each tab has a uniform organization, which, we believe, makes the GUI consistent and comprehensible. With the exceptions of the log tab and the about tab, each tab has **six areas**. Figure 8 shows the taxonomy tab. Let us describe the six areas of this tab.

The **first area** shows the available tabs: Profiles, Shell, TBoxes, ABoxes, Concepts, Roles, Individuals, Assertions, Taxonomy, Role Hierarchy, ABox Graph, Queries, Rules, Log, and About tab. The **second area** is the status display. It displays the current objects, the current namespace, the current profile (representing the current server connection), as well as the current communication status. The clipboard content is not shown, only the cardinality of the sets of selected objects (in the small number fields). The selected objects are highlighted once an appropriate tab is selected. The **third area** shows the history navigator. The **fourth area** is the tab-specific main area. Tab-specific

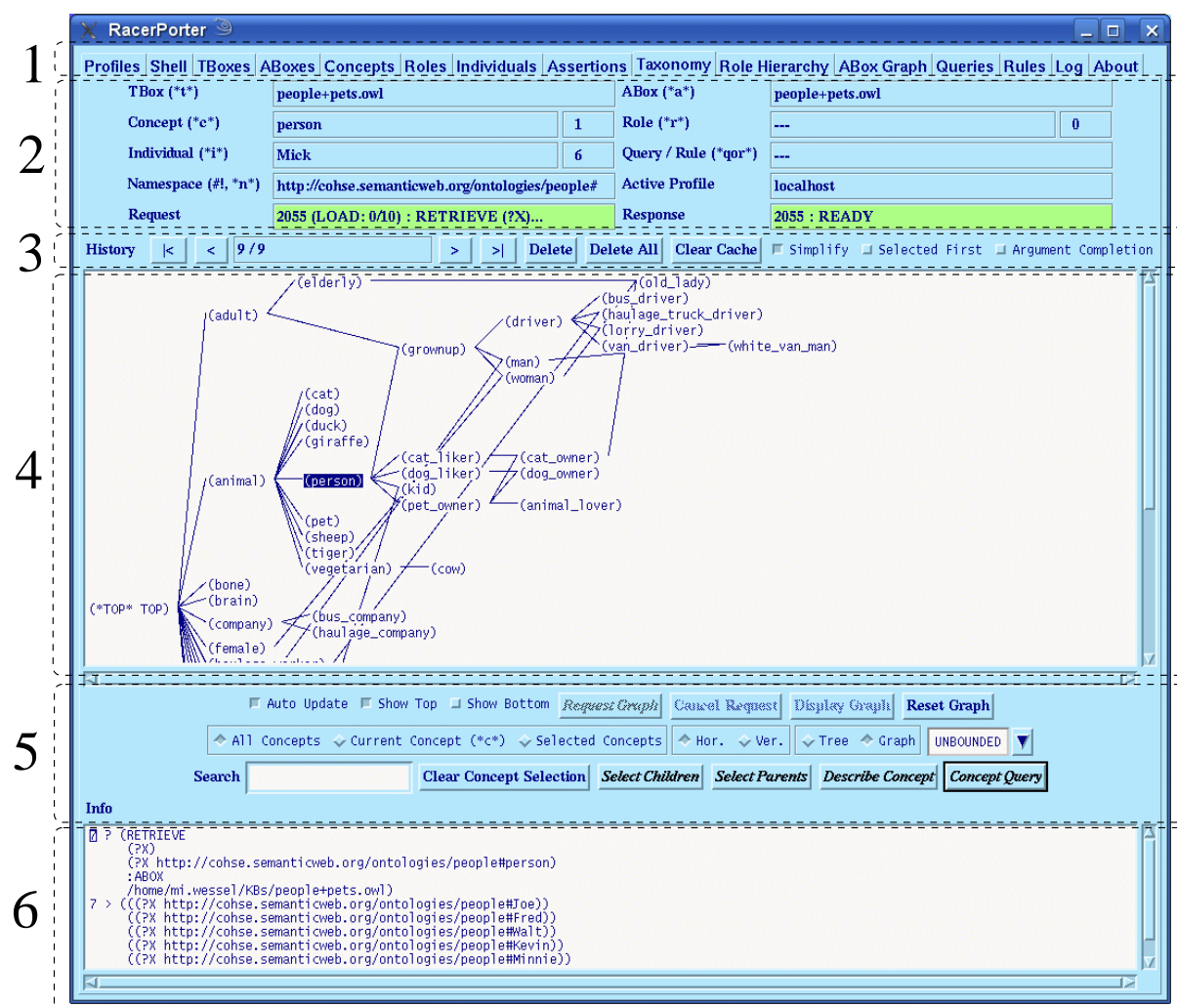


Figure 8: RACERPORTER – The Taxonomy Tab

display options and commands are then presented in the **fifth area**. Finally, there is the info display which is the **sixth area**. The info area is similar to the shell; however, it only “echos” the shell interaction (accepts no input). All user-invoked KRSS commands are put into the shell which are thus also echoed in the info display. This helps to avoid opaqueness, and as a side effect, the user learns the correct KRSS syntax.

The taxonomy and the ABox graph tabs use graph panes for the fourth area. With the exception of the shell, log and logo tab, the other tabs use list panes. List panes allow single or multiple selections of items; selected items represent the selected objects (clipboard). The last selected item specifies the current object. A search field is always present and allows to select objects by name. Selected items will appear on the top of the list if the “Selected First” checkbox is enabled. Some list panes display additional information on their items in multiple columns; e.g., in case of the TBox pane, not only the TBox name is shown, but also the number of concepts in that TBox, profile and DLS server information is shown in the profiles list, etc.

The graph panes are more complicated to handle since they allow to specify focus, layout as well as update options. In case of the ABox graph pane, one can determine which individuals and which edges are displayed. Thus, for both individuals and roles, the focus can be set to the current objects, to the selected objects, or to all objects. Appropriate radio buttons are provided. Additional radio buttons control whether only told role assertions, or also inferred role assertions shall be shown. Additional buttons allow to specify whether the graph display shall be updated automatically if the focus or layout options changes, or whether the user determines when an update is performed. In the latter case, the user first uses the button “Request Graph” to acquire the information from RACERPRO (phase 1). Once the graph is available, the “Display Graph” button becomes enabled; if pushed, the graph layout is computed and displayed. Both phases can be canceled (and different focus and layout options selected subsequently) if they should take too long¹².

Finally, let us briefly discuss some features of the shell. The shell provides automatic command completion (simply press the tab key) as well as argument completion. The potential commands / arguments are presented in a pop-up list. Command completion is achieved by accumulating all results ever returned by KRSS commands. In order to make it easy to reexecute commands, the shell maintains its own shell history (not to be mistaken with the navigation history). Since the shell is tailored for KRSS commands in Lisp-syntax, we provide parenthesis matching, convenient multi-line input as well as pretty printing. Moreover, one no longer has to use full qualified names for OWL resources. To select the objects returned by a shell command, one has to hit the appropriate button (e.g., the “Selected Individuals := Last Result” button).

The log tab keeps a communication log which can be inspected at any time in order to learn what the DLS is currently doing. The current communication with RACERPRO is also visualized in the request and response status fields; appropriate colors are used to visualize the different stages of such a communication (first the request is send, then RACERPRO is busy, then the result is received over the socket, finally the result is parsed, etc.; note that errors can occur at any time in such a processing chain).

RACERPORTER includes an Emacs-compatible editor with buffer evaluation mechanism (see Figure 1). Furthermore, RACERPORTER will provide for visualization of explanations if an inconsistency occurs. This is done in two ways: a) by highlighting of culprits for inconsistency in the axioms and assertions tabs, and b) by highlighting of culprits in the editor window.

In the queries tab, nRQL queries can be executed. Besides of this, next versions of RACERPORTER will also allow for sending SPARQL queries and displaying result tuples in the special SPARQL tab.

RACERPORTER also includes basic functionality to start and stop RACERPRO servers; startup and connection options can be specified with a profile editor. Finally, we want to stress that RACERPORTER is a multi-session tool; thus, the current state and history, but also the shell content, is session or profile specific. Thus, the content of the shell must be saved and reinstalled once the original profile or session is reactivated. As one expects, this can be very memory-intensive, but thats the only way to do it.

¹²Although RACERPORTER does not block in phase 1, unfortunately we have to block the GUI in phase 2 due to a restriction of the GUI framework we are currently using.

6.4 Some Notes About Performance

We learned that a lot of effort must be put into an OBIT until it can be successfully used on large KBs. We tested the redesigned version of RACERPORTER on the OpenCyc ontology [Ope] consisting of 25568 concepts, 9728 roles and 62469 individuals. The result is that RACERPORTER can be used to browse and visualize this ontology. Moreover, time and memory requirements are not too bad. To achieve this, many aspects of the original code had to be reworked thoroughly. This not only concerns the choice of appropriate container data structures “that scale”, but also issues like communication over socket streams. For example, in our case it was no longer possible to simply coerce sequences of characters read from the socket connected to RACERPRO to strings (although this is a very fast operation), since these strings simply get too big to be represented in the environment we use. For the implementation of the clipboard, we originally used lists as data structures. However, if the clipboard contains some ten-thousand instances, one can easily imagine that performance breaks down, since checking whether an object is selected (and thus a member of the clipboard) or not is an operation which has to be performed very frequently. Furthermore, in order to reduce socket communication latency, caches must be used in order to achieve an acceptable performance. Even the design of these caches is demanding.

6.5 Conclusion

Summing up, we have presented design principles for OBITs and showed how they are realized in RACERPORTER. Given the abundance of visualization facilities required, an OBIT has to link interactions and results into a coherent information flow. In RACERPORTER, this information flow is established not only between the tabs and the system core (like a plugin architecture as it is realized, e.g., in Protégé) but also between certain tabs. To be user-friendly, an OBIT must come up with easy browsing and navigation solutions even for large ontologies. Although the existing tools are already very impressive, there is certainly room for enhancements, especially regarding visualization of and navigation in large ontologies in combination with powerful text-based editing techniques.

7 iCom

7.1 Introduction

ICOM (version 3.0) is an advanced CASE tool, which allows the user to design multiple extended ontologies. Each project can be organised into several ontologies, with the possibility to include inter- and intra-ontology constraints. Complete logical reasoning is employed by the tool to verify the specification, infer implicit facts, devise stricter constraints, and manifest any inconsistency. ICOM is fully integrated with a very powerful description logic reasoning server which acts as a background inference engine. The intention behind ICOM is to provide a simple conceptual modelling tool that demonstrates the use of, and stimulates interest in, the novel and powerful *knowledge representation* based technologies for *database* and *ontology* design.

The conceptual modelling language supported by ICOM can express:

- the standard Extended Entity-Relationship data model or the standard UML class diagrams, enriched with disjoint and covering constraints and definitions attached to classes and relations by means of view expressions over other classes and relationships in the ontology;
- inter-ontology mappings, as inclusion and equivalence statements between view expressions involving classes and relationships possibly belonging to different ontologies.

The tool allows for the creation, the editing, the managing, and the storing of several interconnected ontologies, with a user friendly graphical interface. In particular, as analysed in the deliverables D05 and D13, ICOM provides a general framework to support the typical tasks involved in such activities:

- Authoring of concept descriptions: in this task the author wants to add a new concept description to the ontology or modify a concept description that was already contained in the ontology. This may happen either in the design phase of the ontology or during the maintenance phase. After producing a candidate description of the concept, the author needs to understand the implicit consequences of his modelling and the interaction of this description with the other descriptions in the ontology.
- Generating of concept descriptions: in this task the ontology designer wants to add a new concept to the ontology, but finds it difficult to describe it. To obtain a starting point for the concept description, the designer wants to automatically generate an initial description of the new concept that is based on the position of this concept in the subsumption hierarchy.
- Structuring of the ontology: in this task the ontology designer wants to improve the structure of an ontology by inserting inter-mediate concepts into the ontology diagram. He needs support to decide where to add such concepts and how to describe them.

The ICOM tool is written in standard Java 5.0, and it is being used on Linux, Mac, and Windows machines. ICOM communicates via the DIG 1.1 protocol with a description logic server. ICOM provides an interface for exporting ontologies in OWL-DL format, and for importing and exporting ontologies in UML-XMI class diagrams format.

The version 3.0 of the ICOM tool is loosely based on the ICOM tool previously released in 2001 as an Entity-Relationship editor. The foundations of the user-computer interaction have been radically changed according to the experience of the first ICOM and the studies of the first part of the TONES project. The system has been completely re-implemented, using different graphic libraries.

This is the first release of the new ICOM, and it is still not intended to be announced to the outside community. Rather, we intend to start now an experimentation phase within the TONES consortium.

7.2 Optimisations

The ICOM tool is intended as a very general *framework* for ontology design and maintenance, and as such it may include with its design and maintenance workflow some of the technologies as specified in deliverable D13. In particular, the explicit support for lightweight description logics on the one hand, and the very expressive description logics such as OWL 1.1 and description logics with concrete domains is underway. The tool is being currently extended to support explicitly and completely also the task of bottom-up construction of ontologies, by exploiting the techniques for ontology extraction from database schemas described in the deliverable D13.

7.3 How to use

A Linux, MacOSX, or Windows machine is required, with Java 5.0 compatible virtual machine previously installed. ICOM comes as a standalone folder, to be copied anywhere in the hard disk. A Description Logic reasoning server supporting the DIG protocol needs to be installed as well, in order to be able to make deductions. After the installation, you will find an executable file “`ontoeditor`” in the top level directory; execute it (either the `.bat` or `.sh` extension, depending on your platform), and the system will be launched. The “`ontoeditor`” file runs only the editor; it does not start the reasoning component. The reasoner server must be independently launched before or after launching ICOM.

This is a step list for installing and running ICOM:

1. install a Java 5.0 compatible virtual machine (for example Sun JRE 5.0 at http://java.sun.com/javase/downloads/index_jdk5.jsp)
2. install a Description Logic server accepting DIG connections (for example RacerPro at <http://www.racer-systems.com/>)
3. download ICOM executable files from the ICOM home page

<http://www.inf.unibz.it/~franconi/icom/ontoeditor.zip>

4. unzip the file `ontoeditor.zip` into a new directory in the system.
5. execute the Description Logic reasoning server.
6. execute ICOM, by running either the `ontoeditor.sh` file on Linux and MacOS, or the `ontoeditor.bat` file on Windows.

A manual on the operation in ICOM is available in the distribution.

As a quick example, let us consider the concrete example user scenario for ontology design, as presented in deliverable D07; this is available in the ICOM distribution as `design-project.project`. Let us consider the original ontology represented by the diagram in Figure 6 of Section 4.1 of deliverable D07. The ontology states that mobile calls are a kind of calls (IsA link between entities); that phone points are partitioned between cell points and landline points (i.e., any phone point is either a cell or a landline point, but not both: they form a covering and disjoint IsA hierarchy). Each call has at least one

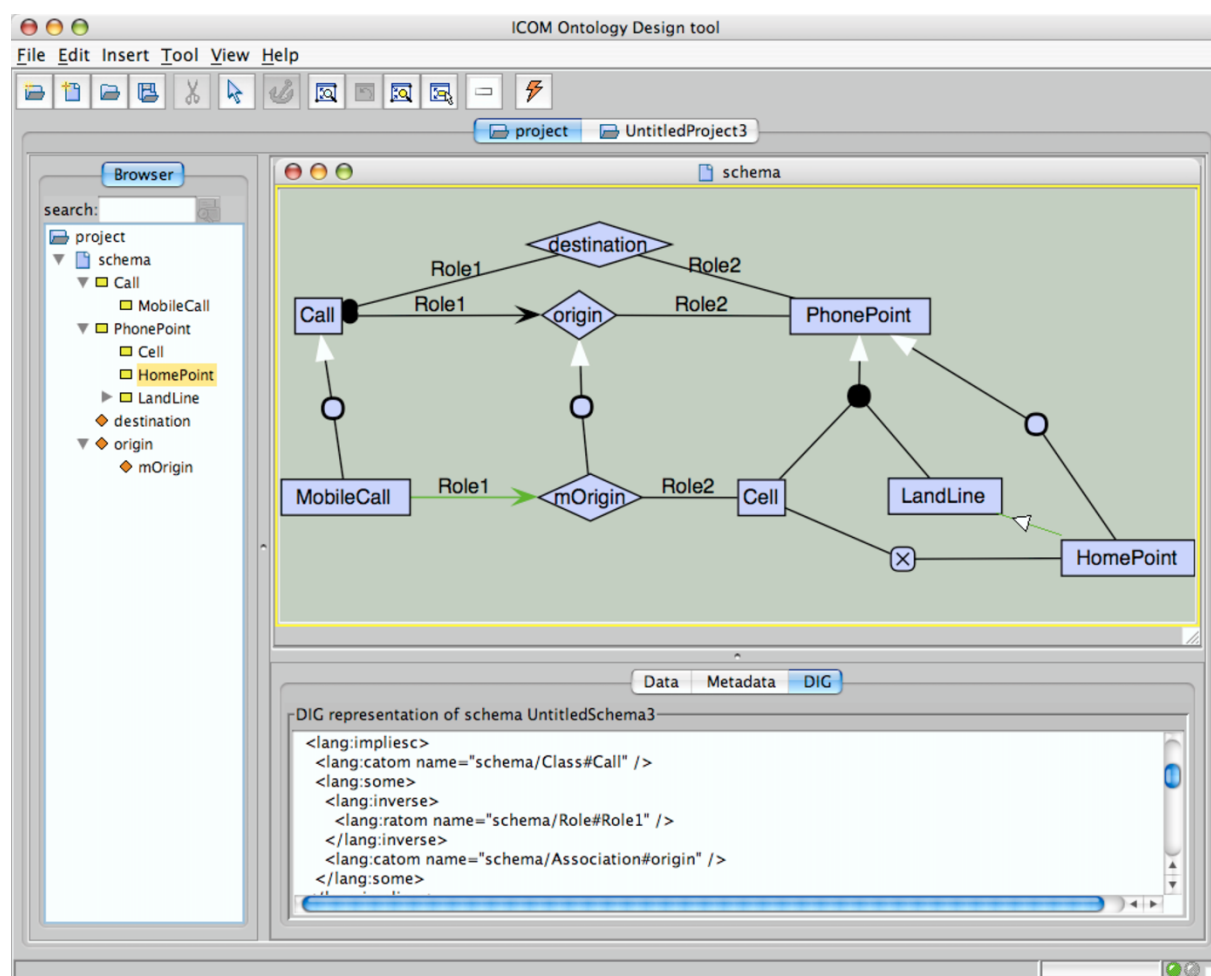


Figure 9: The ontology design scenario in ICOM.

destination phone point (mandatory participation of `cell` to `destination`), while it has exactly one origin phone point. Mobile calls are related to cells through the `mOrigin` relationship. Finally, the binary relationship `mOrigin` is included in the binary relationship `origin`.

Which are the consequences of the above ontology? ICOM is able to automatically complete the diagram in the way depicted in Figure 9. The added constraint (that ICOM shows in green) states that in the above scenario it is necessarily true that each mobile call may have an origin from at most one cell point. The reason why this happens has been explained as follows. The `mOrigin` binary relationship is included in the `origin` binary relationship, i.e., any pair in `mOrigin` is also among the pairs in `origin`. Since each call participates exactly once as first argument to the `origin` relationship, if I take a generic sub class of calls, such as the class of mobile calls, and a sub relationship of the `origin` relationship, such as `mOrigin`, then we can conclude that necessarily each mobile call participates at most once as first argument to the `mOrigin` relationship. Nothing can be concluded about the minimum participation, since the `mOrigin` relationship may not contain all calls in the `origin` relationship.

8 OntoExtract

8.1 Introduction

The Ontology Extraction module is a demo implementation of techniques presented in [ton07] (section 12, “Ontology Extraction from DB Schemas”). Given a relational database, the Ontology Extraction module authors an ontology that is to be used as a conceptual view over the data. The semantic mapping between the database schema and its conceptualisation is captured by associating views over the data source to the elements of the extracted conceptual model. The Ontology Extraction module takes as input an XML file describing the relational schema together with a set of integrity constraints expressed over it. It produces as output the conceptual model described in XML format which, for its graphical representation, is imported in ICOM - the Ontology Design tool (see Section 7).

8.2 How to use

The tool is written on Java, so the only requirement is JRE 1.5. It should work on any platform supporting it.

The Ontology Extraction module comes as a standalone folder, to be copied anywhere in the hard disk. To run the module, execute the following command:

```
java -jar ontoextraction-jar input-file-path output-file-path
```

where

- *ontoextraction-jar* is the full path of `ontoextraction.jar` file,
- *input-file-path* is the full path of an input XML file,
- *output-file-path* is the full path of an output file with the `.project` extension, in order to import it to ICOM tool.

For a graphical representation of the extracted schema, launch ICOM and open the project file generated by the module. In addition to the diagram that can be explored and edited in the main window, SQL view definitions are associated to classes and relationships as metadata that can be viewed in the corresponding tab of the editor.

8.2.1 Input file format

The Ontology Extraction module takes as input the relational schema described in a specific XML format. The XML DTD and Schema files describing the input format are available in the `docs` folders. Moreover, an example schema, together with the resulting ICOM project file, is available in the `example` folder. The `readme.rtf` file describes the structure of the input file.

9 SONIC

The name SONIC stands for “simple ontology non-standard inference component”. This system implements a whole collection of non-standard inferences.

9.1 Introduction

In its current version SONIC implements a range of so-called non-standard inferences. SONIC comprises basically two parts. One is the SONIC reasoner, which implements the non-standard inferences. The other part is ontology editor component that realises a graphical user interface to access the inferences in an easy way. We will concentrate in this deliverable and report on those inferences that are helpful in realizing ontology design and maintenance tasks as described in the TONES deliverable D05. In particular the SONIC system can support naive users in assisting in the following ontology design and maintenance tasks – as identified in the TONES deliverable D05:

Generating Concept Descriptions. The ontology designer wants to add a new concept to the ontology, but finds it difficult to describe it. To obtain a starting point for the concept description, the designer wants to automatically generate an initial description of the new concept that is based on the position of this concept in the subsumption hierarchy.

Structuring the Ontology. The ontology designer wants to improve the structure of an ontology by inserting intermediate concepts into the subsumption hierarchy. He needs support to decide where to add such concepts and how to describe them.

Bottom-up Construct. The ontology designer wants to design the ontology bottom-up, i.e., by proceeding from the most specific concepts to the most general ones. This should be supported by automatically generating concept descriptions from descriptions of typical instances of the new concept.

Ontology Customization. An ontology user wants to adapt an existing ontology to her purposes by making simple modifications. Since she is not an expert in ontology languages, she works with a simpler language than the one used to formulate the ontology and/or with graphical frame-like interfaces.

Concept Inspection. The ontology designer wants to display a concept description in a way that facilitates understanding of the concept’s meaning.

The first three of these ontology tasks can at least be partially realized by computing the *least common subsumer* (lcs). This technique has been discussed in the TONES deliverable D13 [ton07]. SONIC implements this inference for unfoldable \mathcal{ALN} -TBoxes and its sub-languages. The user can load such a TBox into PROTÉGÉ, classify it and then pick a collection of concept names that she wants to introduce a new concept name for. The lcs of the selected concepts is computed by the SONIC reasoner and displayed to the user in the GUI part, where the user can edit the returned concept description, assign it a concept name and add the new concept definition to the TBox.

The task of structuring the ontology is especially well supported in SONIC by another service that is based on the lcs inference. In order to provide more information on the choice of a collection of concepts whose lcs would introduce a new node in the concept

hierarchy, SONIC implements the computation of a concept lattice of concept descriptions. More precisely, it implements the hierarchy of least common subsumers of the power set of a set of selected concepts from the ontology. Thus the concept hierarchy of these lcs concept descriptions is displayed to the user. She can now see whether one of the selected concept descriptions is causing a too general lcs and leave this concept out in the computation of the lcs concept description. The method for the computation of the hierarchy (w.r.t. subsumption) of the prospective lcs concept descriptions without computing these concept descriptions themselves was introduced in [BM00]. The method is based on *formal concept analysis*, see [GW99]. SONIC implements this method for the DL *ALCN*.

Ontology customization is another ontology design and maintenance task that SONIC facilitates. The system implements support for the customization of an expressive *background ontology* by a less expressive *user ontology*. The approach of computing non-standard inferences w.r.t. a background ontology was introduced in [BST04]. Here the user builds a user ontology in the user DL by extending an expressive background ontology \mathcal{T} written in an expressive DL and by using the names from the signature of the background ontology. Thus, the user DL is extended by the use of concept names that stand for complex concept descriptions expressed in the more expressive background DL in \mathcal{T} . SONIC implements the relaxed notion for computing the commonalities of a collection of concept descriptions – namely different forms of *good common subsumers* (GCS). A GCS is a common subsumer, but does not need to be the least one. In [BST07] different algorithms for computing a good common subsumer were proposed. SONIC offers the subsumption based computation of GCS defined in [BST07], which showed a good behaviour in practice. However, SONIC does not yet support the use of several ontologies, but the background and user ontology have to be in the same file, in order to be used.

The third ontology design and maintenance task SONIC supports is concept inspection. Two inference services implemented in SONIC facilitate concept inspection. One is *concept approximation*, which “translates” concept descriptions written in an expressive DL into a less expressive DL. For example to suite the display of a frame-based ontology editor or simply to support inexperienced users when exploring a knowledge base a simpler DL might be desirable. Concept approximation generalizes the concept description only as little as possible. For references on this non-standard inference refer to Deliverable 13 of the TONES project [ton07].

Since the concept description returned by approximation or also the lcs can grow very large in practice, it is necessary to display the approximated concept description in a succinct way. In these cases a *minimal rewriting* of the concept description is computed. A minimal rewriting is a concept description equivalent to the original concept description of smaller, more precisely of minimal concept size. These kind of rewritings (re-)introduce concept names from the TBox for sub-concept descriptions in the concept description and thus are more succinct. SONIC implements a heuristic for computing a small, but not in necessarily minimal rewriting. The method was introduced for unfoldable ontologies in *ALC* in [BKM00].

The development of the SONIC system started in a DFG project (Grant BA 1122/4-3). In its first published version SONIC supported mainly the computation of the lcs in the

DL \mathcal{ALN} (and its sub-languages) and concept approximation of \mathcal{ALC} - by \mathcal{ALN} -concept descriptions, see [TK04b, TK04a]. This version of SONIC was equipped with a GUI counter part that could be used with the OILED ontology editor [BHGS01]. In a second phase the development of SONIC was partially supported by the Network of Excellence for Semantic Interoperability and Data Mining in Biomedicine (NoE 507505). During that phase SONIC was extended to the ontology editor PROTÉGÉ and by the non-standard inferences minimal rewriting and a version of the good common subsumer, see [Tur05]. Currently, SONIC supports only the ontology editor PROTÉGÉ.

Recently, during the TONES project SONIC was extended by the above mentioned computation of lcs lattices. Furthermore the older implementations were improved for better performance, which is an on-going process. Furthermore SONIC is currently being extended by a DIG 2.0 interface. Since SONIC needs the connection to a standard DL reasoner, SONIC will implement a DIG 2.0 standard client, see [TBK⁺06]. Since SONIC does not keep a copy of the knowledge base, it needs access to *told information* from the DL reasoner and, of course, SONIC will provide an implementation of the NSI extension of DIG. Both extensions of the core DIG 2.0 interface were also described in [TBK⁺06].

As most DL systems, SONIC is under development. Next, besides the improvement on performance, we plan to implement more inferences for the customization scenario. More information on SONIC can be obtained from the web-page <http://lat.inf.tu-dresden.de/systems/sonic.html>.

9.2 Optimizations

Regarding the optimizations in SONIC one must bear in mind that the inferences implemented in this system are computation problems and the output of most of the algorithms can grow very large—in case of approximation the output can grow up to double exponential in size of the input. Thus the optimization for these kind of inferences are expedient.

The algorithm that SONIC implements operate on concept descriptions. Thus, if defined w.r.t. an (unfoldable) TBox, a concept name has to be unfolded in order to apply the inference – a process that is well-known to blow-up the concept description exponentially. Moreover, the NSIs lcs, gcs and approximation require that the concept description is in a certain normal form. These normal forms can again generate an exponential blow-up of the unfolded concept descriptions. So, SONIC employs *lazy unfolding* and *lazy normalization*, i.e., concepts are not unfolded and normalized completely at the beginning of the computation, but these steps are interlaced with the generation of the result concept description. More precisely, first only the top-role level of the concept description is unfolded and normalized, the subsequent one is only unfolded and normalized, if necessary during the computations.

The approximation algorithm is needed not only to support the design and maintenance scenarios directly, but also to “translate” concept descriptions in order to obtain concept descriptions for which other NSIs are available or where the application of the NSI directly is only of limited usefulness. In case of DLs that provide disjunction the lcs of concepts written in this DL is simply their disjunction. In such a case the user does not learn anything about the commonalities. The idea is to first approximate a concept and then to apply the NSI to the approximated concept. Thus we in particular implemented

ways optimizing the computation of concept approximation. One way to do this is to reduce the number of recursive calls generated for the computation of existential restrictions. During the computation of the cross-product some recursive calls can be avoided since they will yield existential restriction more general (and thus redundant) to others. A second, promising way implemented in SONIC to optimize approximation is to split the approximation of a concept description at the conjunction. So, instead of computing $\mathbf{approx}(C_1 \sqcap C_2)$ we compute $\mathbf{approx}(C_1) \sqcap \mathbf{approx}(C_2)$. Now, if C consists of two conjuncts of size n then the approximation of C takes some $a^{b^{2n}}$ steps while the conjunct-wise approach would just take $2a^{b^n}$. Unfortunately, this method does not yield correct results for arbitrary concept descriptions. The approach is only applicable to concept descriptions that obey certain syntactic conditions, see [TB07]. For this kind of so-called *nice* concept descriptions the approach yields a considerable speed-up.

In one of the next releases of SONIC, we plan to implement caching for the inferences in SONIC, so that results can be obtained in subsequent computations. For concept approximation again nice concept descriptions are a prerequisite for a non-naive use of caching. Say, we want to approximate $C \sqcap D$, where C and D are complex \mathcal{ALCN} -concept descriptions. Now, if we already have cached the approximation of C and $C \sqcap D$ is nice, we only need to compute the approximation of D and conjoin it with the cached result. However, we expect to improve the performance of SONIC in the near future.

9.3 How to use

The SONIC distribution contains the mainly following:

- `Read-me.txt`,
- `install-sonic.sh` for Allegro Common Lisp,
- `/bin/` SONIC executables for Allegro Common Lisp,
- `SonicPlugin.jar`

Before installing SONIC, make sure that the following resources and software components are installed on your system.

Requirements

We recommend a x86 i686 processor and at least 128 RAM to run SONIC. So far SONIC can only run under Linux. Moreover SONIC requires that Java (for example Suns J2SE 1.5 Version 2.1 or higher) is installed on your system. In addition one of the supported Lisp systems must be installed. Currently SONIC can run under: Allegro Common Lisp (version 7.0 or 8.0). Besides installations of this LISP system and Java you need two software components to be installed on your system to run SONIC.

RACER SONIC uses RACER as a background reasoner, thus RACER must be installed on your system before you can run SONIC. There are two RACER system components needed to run SONIC— the RACER server and LRACER. While the LRACER component

comes with the SONIC distribution and is installed by the SONIC install script, the RACER server must be down-loaded and installed by you. SONIC uses RACER server version 1-8 or higher. We recommend to use the latest available version. The RACER server can be freely down-loaded for research purposes from www.racer-systems.com

PROTÉGÉ SONIC comes with a plug-in for the ontology editor PROTÉGÉ. Currently, SONIC supports PROTÉGÉ version 3.2, which can be downloaded from <http://protege.stanford.edu/download/>.

Installation

SONIC comes with an installation script called `install-sonic-acl.sh` for Allegro Common Lisp. Under Linux operating system you can start the installation script by executing `install-sonic.sh` in the shell. This script installs SONIC and LRACER. It edits PROTÉGÉ's preferences s.t. the SONIC panels are loaded when PROTÉGÉ is started.

In case the installation procedure is not successful, please refer to the file `Read-me.txt` from the SONIC distribution for trouble shooting.

Getting started

SONIC consists of two parts: the one part realizes the graphical user interface as a plug-in for PROTÉGÉ and the other part implements the non-standard inferences in Lisp. In the current implementation the reasoning component of SONIC can be started directly as an executable. The SONIC plug-in for PROTÉGÉ is loaded when PROTÉGÉ is started, if SONIC is correctly installed as described above. Moreover one has to start the RACER server before starting SONIC. In the current version all three components, the RACER server, PROTÉGÉ and the SONIC script have to run on the same host. To sum it up the complete start procedure for SONIC is:

1. Start RACER (with port 8080) in a shell.
2. Execute `sonic`
3. Start PROTÉGÉ as usual, see PROTÉGÉ manual.
4. Connect PROTÉGÉ to the DIG reasoner RACER.
5. activate the SONIC panels via `configure` in the OWL Project menu.

After this the LCSPanel and the ApproxPanel are active. On the ApproxPanel you can select a concept from your ontology and compute its approximation. Similarly, you can select a group of concepts from your ontology on the LCSPanel and compute their least common subsumer. From both ontologies the resulting concept descriptions can be edited and then saved to the ontology.

10 InstExp

10.1 Introduction

INSTEXP (Instance Explorer), is an interactive tool that aims to support enriching an ontology by asking questions to a domain expert. It asks questions of the form “*Is it true that objects that have properties a, b, c also have the properties d, e, f ?*”. The domain expert is expected to answer “yes” or “no”. If she answers with “no”, then she is expected to provide a counterexample, and this counterexample is added to the ontology. If she answers “yes”, then the ontology is updated with a new inclusion axiom. When the process stops, the ontology is complete in a certain sense. The advantage of the method is that it guarantees to ask the minimum number of questions to the expert in order to acquire the missing part of the knowledge. The theoretical background of INSTEXP was explained in detail in [ton07, BGSS06, BGSS07]. INSTEXP addresses the tasks of structuring an ontology, which was mentioned in [ton07].

The development of INSTEXP was partially supported by the EU projects TONES (IST-2005-7603 FET) and Semantic Mining (NoE 507505), and the German Research Foundation DFG (GRK 334/3).

Some information on INSTEXP and the distribution can be found under <http://wwwtcs.inf.tu-dresden.de/~sertkaya/InstExp>

10.2 Optimizations

INSTEXP implements an extension of the attribute exploration algorithm [Gan84, GW99] developed in the field of Formal Concept Analysis. The extension of attribute exploration for use in completion of Description Logics ontologies was described in [BGSS06, BGSS07].

The completion algorithm guarantees to ask the minimum number of questions that have positive answer. The number of questions that have negative answers depends on the counterexamples provided by the expert. Theoretically, there is a set of counterexamples that lead to the minimum number of questions with negative answers. However, in a real world application one can not always expect that the expert is able to provide this set of counterexamples.

The completion algorithm keeps a list of implications, and often needs to compute closure under this set of implications. For computing implicational closure, we have implemented the efficient closure algorithm *linclosure* described in [Mai83]. For representing sets of so called attributes, we have used bit vectors for efficient set operations.

10.3 Usage

INSTEXP is implemented in the Java programming language as an extension to the Swoop [SP04] ontology editor (see also Section 5). It runs on any platform that has Java Runtime Environment (JRE) version at least 1.5.0. JRE can be downloaded from <http://java.sun.com>.

INSTEXP can be obtained from <http://wwwtcs.inf.tu-dresden.de/~sertkaya/InstExp/swoopInstExp.tgz>. The zip package contains version v2.3 beta 3 of the Swoop

ontology editor patched with INSTEXP . You can start the patched Swoop just like you start Swoop as usual. If you are using a UNIX/Linux operating system,

1. Untar the package using the command: `tar zxvf swoopInstExp.tgz`. It will produce a directory with name `swoopInstExp`.
2. change to the `swoopInstExp` directory using the command: `cd swoopInstExp`.
3. set the execute permission of the start up script: `chmod +x runme.sh`
4. run the start up script: `./runme.sh &`

If you are using Windows, unzip the package and execute the `runme.bat` file in the `swoopInstExp` that will be produced.

Starting InstExp : In the Swoop window that will open, load the ontology you want to work with, and classify it by selecting Pellet in the reasoner combo box. Now start INSTEXP from the *Advanced* menu by selecting *Instance Explorer*. INSTEXP will start in a separate window. In the scroll box on the left of the window, you will see the class hierarchy of the selected ontology. From the hierarchy, you can select the concepts you want to include in the completion process. You can either add concepts one by one by selecting a concept and clicking the *Add* button, or you can add all subclasses of the selected concept by clicking the *Add subclasses* button (If the selected concept does not have any subclasses, no concept will be added). When you click the *Add* or *Add Subclasses* button, in *Context* tab on the right of the window, a table will occur. The column(s) of the table consist(s) of the concept(s) you have added, and the rows consist of the instances of the added concept(s). A “+” in row x and column y of the table means that individual x is an instance of the concept y . Similarly, a “-” means that x is an instance of $\neg y$, and a “?” means that x is neither an instance of y , nor an instance of $\neg y$. In case you accidentally add a concept you did not really mean to add, you can click on the *Reset* button. It will remove all the concepts added until then together with their instances, i.e., the *Context* table on the right will be resetted. Once you finish adding the concepts you want to, click on the *Start* button in order to start the completion process.

Once you hit the *Start* button, the interactive completion process will start. In the *Console* at the bottom of the window, INSTEXP will start asking you questions of the form:

"Is it true that the objects satisfying properties L also satisfy the properties R ?"

where L and R are subsets of the concept names you have added.

Confirming a question: If this is the case in your application domain, i.e., if every individual that is an instance of all of the concepts in L is also an instance of all of the concepts in R , then you confirm the question by clicking the *Yes* button in the *Console* tab at the bottom of the window. In this case, the inclusion axiom $\sqcap L \sqsubseteq \sqcap R$ is going to be added to your ontology, where $\sqcap L$ stands for $\sqcap \{C_i \mid C_i \in L\}$. The updated ontology is going to be reclassified and INSTEXP will come up with a new question.

Rejecting a question: In case the question does not hold in your application domain, i.e., if there is an individual that is an instance of all of the concepts in L but is an instance

of $\neg C_i$ for at least one of the concepts in R , then you reject the question by clicking the *No* button. In this case the *Counterexample editor* tab on the right, and the *Messages* tab in the bottom will become active. At this moment, INSTEXP is waiting you to provide a counterexample.

Providing a counterexample: In order to help you in generating a counterexample, the *Counterexample editor* tab will show you the individuals in your ontology that might potentially be turned into a counterexample to the question you have rejected. They are individuals in your ontology that are instances of $\sqcap L$, but that are not instances of at least one concept appearing in R . If there are no such individuals in your ontology, the *Counterexample editor* will only display an individual with name *defaultName*, that is an instance of all of the concepts in $\sqcap L$, but that is not an instance of the concepts in R .

The individuals displayed in the *Counterexample editor* are editable. It means, you can click on the “?” signs and turn them into “+” “-”. As you edit an individual, the *Messages* tab at the bottom will show you basic tips on how to make the selected individual a valid counterexample. It will tell you for which concepts the selected individual should contain a “+” and for which concepts a “-” so that the selected individual becomes a valid counterexample. As soon as the selected individual becomes a valid counterexample, it will be highlighted with green, and the *Ready* button will become active. At this point you can hit *Ready* and provide your counterexample. But you can also continue editing your counterexample to make it more specific and then hit *Ready*. Then your counterexample is going to be added to the ontology and the next question will come.

While you are editing a counterexample, in case you make the selected individual inconsistent, it will be highlighted with red, and a warning message will appear in the *Messages* tab. In this case the *Ready* button will become inactive and you will not be allowed to provide this inconsistent individual.

As mentioned above, in the *Counterexample editor*, the last individual with name *defaultName* is always a new one. It is not one of the individuals already appearing in the ontology. For the ease of use, it is already marked as instances of the classes on the lefthand side of the question. The name of this individual is also editable. You can edit the string *defaultName* and name this individual with a new name. The new name needs to be distinct from existing individual names, otherwise you will get a warning and be asked to provide another name.

When you are ready with preparing a valid counterexample and click the *Ready* button, the counterexample you have provided is going to be added to your ontology, *Context* and *Console* tabs will become active, and *Counterexample editor* and *Messages* tabs will become inactive. The *Context* tab will now display your counterexample as well, and the *Console* tab will display the new question. This will iterate until you have answered all of the questions asked, i.e., until your ontology is complete w.r.t. the relations between the concepts you have added at the beginning. In this case, a small window will pop up and notify the end of completion process. You can close the INSTEXP window, or hit *Reset* and start a new completion process. After the completion, you will notice that the changes you have made to your ontology already appear in the Swoop window and you can further work with Swoop on the enriched ontology. If you want to save your enriched ontology, use the usual *Save* procedure of Swoop under the *File* menu.

Figure 10 shows a screen shot of the INSTEXP window waiting for the user answer

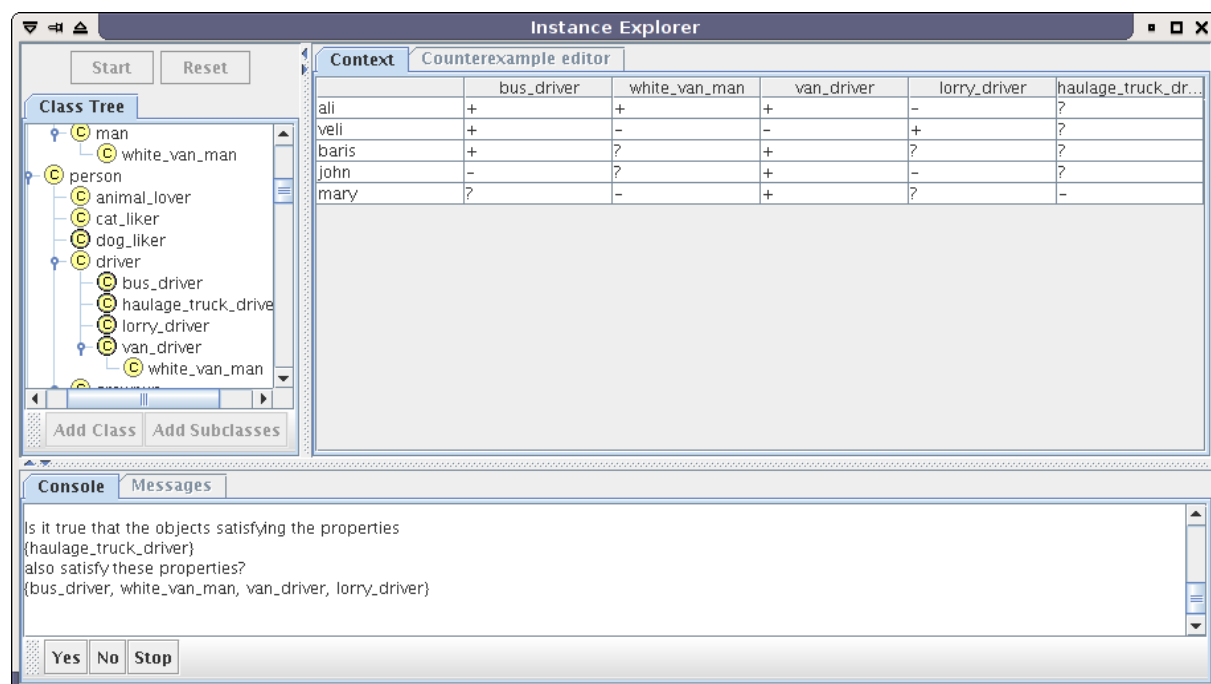


Figure 10: INSTEXP window with question

the question displayed in the *Console* tab. At this moment the user is expected to answer by hitting *Yes*, *No* or *Stop* buttons. When the *Stop* button is hit, the completion process stops, but the changes made until then will NOT be discarded. You will still see the changes in the snoop window. And if you save the ontology in the snoop window, the changes made in the INSTEXP window will be saved to your ontology. Figure 11 shows a screen shot of preparing a counterexample. In the figure, the individual named *baris* is being edited. The *Messages* tab says that the current description of the individual is sufficient for being a counterexample and the *Counterexample editor* highlights the edited individual with green. The ontology used in these screen shots can be found in the distribution under the directory `instexpExample` with name `sane_cows.owl`. As the name of the file also suggests, it is an OWL ontology.

11 DL²RL

11.1 Introduction

In [ton07], we argued that modern model finding tools such as Alloy Analyzer (and his successors) can be very useful for ontology design tasks, since they enable generating and inspecting all (not just canonical) model structures in a systematical way.

In order to adopt a particular finite model finder for ontology design tasks, we defined a set of translation rules for *ALCN* formulas into the Relational Logic underlying the Alloy Analyzer. We tested the applicability of our approach for some use cases using the RACERPRO reasoner and Alloy Analyzer 4.0 [Jac06]. Furthermore, we showed that also *SHIQ* or even *SROIQ* formulas can be successfully translated into Alloy.

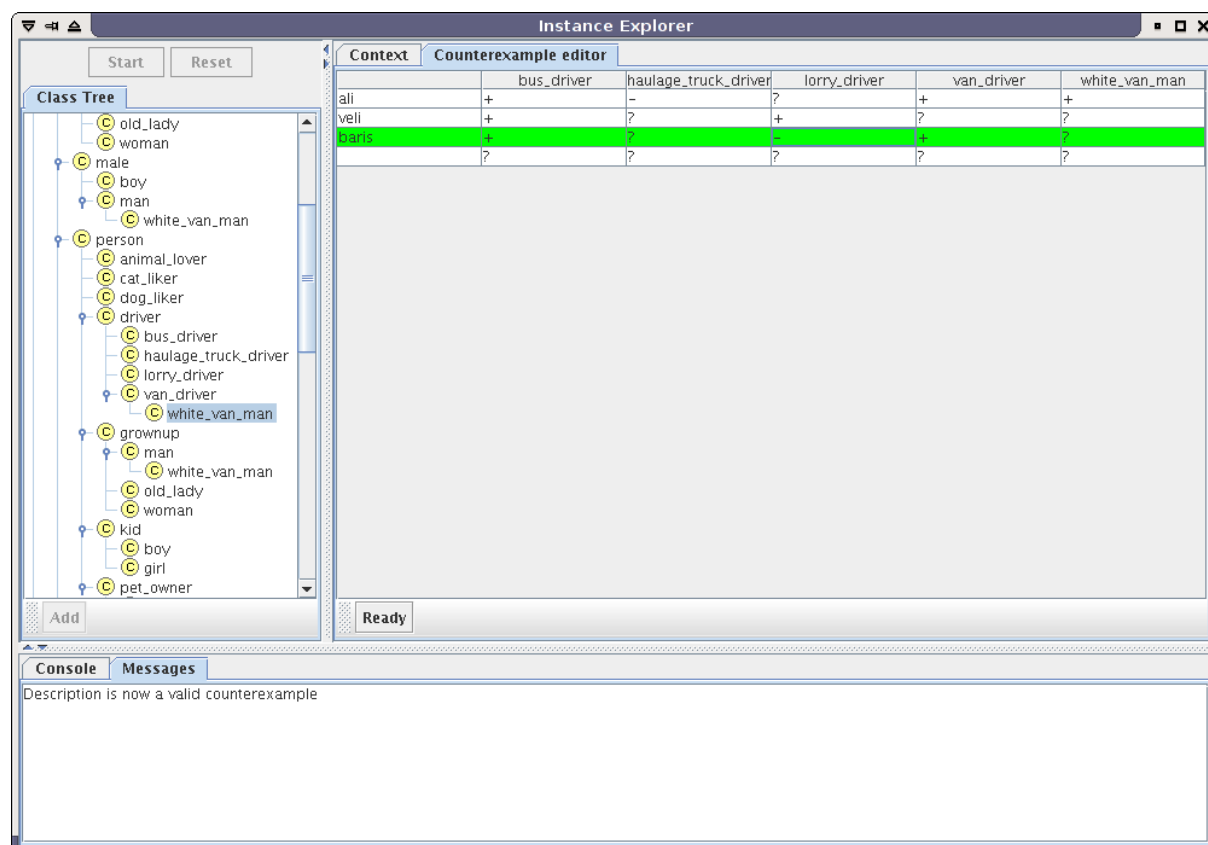


Figure 11: INSTEXP window while preparing a counterexample

The translation algorithm from \mathcal{ALCN} into Alloy's Relational Logic is the core of the DL²RL translation tool being work in progress. The algorithm has been implemented using a Java library, consisting of the packages *dlAST*, *parser*, and *utilities*. The translation service may be invoked with two different kinds of input:

- a pair $\langle TBox, ABox \rangle$
- text file in *.racer* format

The output is first obtained as an in-memory string containing the Alloy specification, which can then be serialized to disk or further processed. These alternatives are provided for better supporting usage of the tool in different scenarios. For example, in case \mathcal{ALCN} input is available in XML format, parsing using a freely available library can be followed by instantiating the Abstract Syntax Trees (ASTs) built for input TBox and a (possibly empty) ABox. From there, translation may proceed. Detailed usage instructions are given in the next section.

11.2 DL²RL – How to Use

In order to invoke the DL²RL translation algorithm taking as input a *.racer* file, an `java.io.InputStream` is to be obtained (for an example see the following Listing and

passed as argument to the constructor for `parser.alcn`, the Java class that realizes the parser.

```
public static String getAlloySpecForDotRacer(String folder,
String filenameWithExtension) {
    String filePath = folder + "/" + filenameWithExtension;
    log("translating .racer into .als for " + filePath);
    InputStream is = null;
    try {
        is = new FileInputStream(filePath);
    }
    catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    alcn parser = new alcn(is); // System.in
    TBox t = null;
    try {
        t = parser.start();
    }
    catch (ParseException e) {
        e.printStackTrace();
    }
    ABox a = t.getAbs().iterator().next();
    String res = a.toAlloy3();
    return res;
}
```

As part of parsing, syntax-directed actions instantiate the Java classes that stand for \mathcal{ALCN} concepts. Parsing is invoked by calling the start rule on the parser (`parser.start()` in the example below). This method returns an instance of `dLAST.TBox`, from which a `Set<ABox>` can be obtained, by invoking method `dLAST.TBox#getAbs()`.

In case a pair `TBox` and `ABox` has been built by other means (other than parsing a `.racer` input file), the functionality of generating the Alloy specification can be accessed by invoking method `ABox#toAlloy3()`. This method returns the string representation (concrete syntax) of the corresponding Alloy specification.

Please notice that only recently the SAT engines used internally by Alloy have been made available as separate components. Moreover, their interfaces account for direct communication with other components (e.g., a slightly modified version of our translation library) thus skipping the cumbersome disk serialization. Those engines are available under following addresses (websites contain publications and documentation besides the software packages themselves):

- Kodkod, <http://web.mit.edu/emina/www/kodkod.html>
- SAT4J, <http://www.sat4j.org/>

Another recent addition positively impacting the usability of Alloy is an Eclipse plugin, <http://code.google.com/p/alloy4eclipse/>, also developed by the Alloy team, with improved interactive syntax-error reporting. The team support facilities such as structural comparison and versioning available from Eclipse, also simplify collaboration while evolving Alloy specs .

11.3 Further Work and Optimizations

Concluding from empirical experiments we performed with the proposed tool, modern highly-optimized tableau-based provers far outperform model finders such as Alloy Analyzer. However, in order to improve the usefulness of tableau-based reasoners for ontology design tasks, it may be a good idea to equip them with model generation capacities like those provided by model finders for identifying unintended models. In the other direction, namely for increasing performance of model finders, DL prover techniques might also be helpful (given the expressivity of the input formulas is in the DL fragment). A tableau prover could be used for satisfiability checking. If there exists a model, the tableau describes a canonical model, which could be further modified in order to derive all models in the sense of model finders. If the expressivity is too high, models might be infinite, however, so the details of this idea have to be investigated carefully.

Increasing performance of constraint-solving engines used by model finders is another crucial requirement for integrating model finders in practical applications like ontology design tools. Recent investigations concern development of faster SAT solvers algorithms and systems (see, e.g., [TJ06]).

References

- [Ali06] Alissa Kaplunova and Ralf Möller. DIG 2.0 Proposal for a Query Interface. 2006. <http://www.sts.tu-harburg.de/~al.kaplunova/dig2-query-interface.html>.
- [BBL05] F. Baader, S. Brandt, and C. Lutz. Pushing the \mathcal{EL} envelope. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, Edinburgh, UK, 2005. Morgan-Kaufmann Publishers.
- [Ber06] Bernardo Cuenca Grau, Boris Motik, and Peter Patel-Schneider. OWL1.1. 2006. http://owl1_1.cs.manchester.ac.uk/xml_syntax.html.
- [BFH⁺94] Franz Baader, Enrico Franconi, Bernhard Hollunder, Bernhard Nebel, and Hans-Jürgen Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.
- [BGSS06] Franz Baader, Bernhard Ganter, Ulrike Sattler, and Baris Sertkaya. Completing description logic knowledge bases using formal concept analysis. LTCS-Report LTCS-06-02, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany, 2006. See <http://lat.inf.tu-dresden.de/research/reports.html>.
- [BGSS07] Franz Baader, Bernhard Ganter, Ulrike Sattler, and Baris Sertkaya. Completing description logic knowledge bases using formal concept analysis. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*. AAAI Press, 2007.

- [BHGS01] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a Reasonable Ontology Editor for the Semantic Web. In *Proc. of the Joint German/Austrian Conf. on Artificial Intelligence (KI 2001)*, volume 2174 of *Lecture Notes in Artificial Intelligence*, pages 396–408, Vienna, Sep 2001. Springer. OilEd download page <http://oiled.man.ac.uk>.
- [BKM00] F. Baader, R. Küsters, and R. Molitor. Rewriting concepts using terminologies. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000)*, pages 297–308, San Francisco, CA, 2000. Morgan Kaufmann Publishers.
- [BLS05] F. Baader, C. Lutz, and B. Suntisrivaraporn. Is tractable reasoning in extensions of the description logic \mathcal{EL} useful in practice? In *Proceedings of the 2005 International Workshop on Methods for Modalities (M4M-05)*, 2005.
- [BLS06a] F. Baader, C. Lutz, and B. Suntisrivaraporn. CEL—a polynomial-time reasoner for life science ontologies. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR'06)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 287–291. Springer-Verlag, 2006.
- [BLS06b] F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in \mathcal{EL}^+ . In *Proceedings of the 2006 International Workshop on Description Logics (DL2006)*, CEUR-WS, 2006.
- [BM00] F. Baader and R. Molitor. Building and structuring description logic knowledge bases using least common subsumers and concept analysis. In B. Ganter and G. Mineau, editors, *Proc. of the 8th Int. Conf. on Conceptual Structures (ICCS'00)*, volume 1867 of *Lecture Notes in Artificial Intelligence*, pages 290–303. SV, 2000.
- [BST04] F. Baader, B. Sertkaya, and A.-Y. Turhan. Computing the least common subsumer w.r.t. a background terminology. In J.J. Alferes and J.A. Leite, editors, *Proc. of the 9th Eur. Conference on Logics in Artificial Intelligence (JELIA 2004)*, volume 3229 of *Lecture Notes in Computer Science*, pages 400–412, Lisbon, Portugal, 2004. Springer.
- [BST07] F. Baader, B. Sertkaya, and A.-Y. Turhan. Computing the least common subsumer w.r.t. a background terminology. *J. of Applied Logic*, 2007. To be published.
- [CRP+93] R. Cote, D. Rothwell, J Palotay, R. Beckett, and L. Brochu. The systematized nomenclature of human and veterinary medicine. Technical report, SNOMED International, Northfield, IL: College of American Pathologists, 1993.
- [DG84] W. F. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

- [Gan84] Bernhard Ganter. Two basic algorithms in concept analysis. Technical Report Preprint-Nr. 831, Technische Hochschule Darmstadt, Darmstadt, Germany, 1984.
- [GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin, Germany, 1999.
- [HM01a] V. Haarslev and R. Möller. RACER System Description. In *Int. Joint Conference on Automated Reasoning*, 2001.
- [HM01b] V. Haarslev and R. Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, 2001.
- [Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
- [KMR04] Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In *Description Logics*, 2004.
- [KPS⁺05] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo C. Grau, and James Hendler. Swoop: A web ontology editing browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144–153, June 2005.
- [LBF⁺06] C. Lutz, F. Baader, E. Franconi, D. Lembo, R. Möller, R. Rosati, U. Sattler, B. Suntisrivaraporn, and S. Tessaris. Reasoning Support for Ontology Design. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *In Proceedings of the second international workshop OWL: Experiences and Directions*, November 2006.
- [LN05] Thorsten Liebig and Olaf Noppens. ONTOTRACK: A semantic approach for ontology authoring. *Journal of Web Semantics*, 3(2-3):116–131, October 2005.
- [LN06] Thorsten Liebig and Olaf Noppens. Interactive Visualization of Large OWL Instance Sets. In *Proc. of the Third Int. Semantic Web User Interaction Workshop (SWUI 2006)*, Athens, GA, USA, November 2006.
- [Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, Maryland, 1983.
- [Ope] OpenCyc. <http://www.opencyc.org/>.
- [PSS93] P. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification from the krss group of the arpa knowledge sharing effort. Technical report, DARPA Knowledge Representation System Specification (KRSS) Group of the Knowledge Sharing Initiative, 1993.
- [Rac] RacerPro, an OWL reasoner and inference server for the Semantic Web. <http://www.racer-systems.com/>.

- [RH97] Alan Rector and Ian Horrocks. Experience building a large, re-usable medical ontology using a description logic with transitivity and concept inclusions. In *Proceedings of the Workshop on Ontological Engineering, AAAI Spring Symposium (AAAI'97)*, Stanford, CA, 1997. AAAI Press.
- [Sea06] Sean Bechhofer. DIG 2.0: The DIG Description Logic Interface. 2006. <http://dig.cs.manchester.ac.uk>.
- [SK06] Ferdinando Villa Sergey Krivov, Rich Williams. Growl, visual browser and editor for owl ontologies. *Journal of Web Semantics*, 2006.
- [SP04] Evren Sirin and Bijan Parsia. Pellet: An OWL DL reasoner. In *Proceedings of the 2004 International Workshop on Description Logics (DL2004)*, volume 104 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
- [Sun05a] B. Suntisrivaraporn. CEL—the manual. Available at <http://lat.inf.tu-dresden.de/systems/cel>, 2005.
- [Sun05b] B. Suntisrivaraporn. Optimization and implementation of subsumption algorithms for the description logic \mathcal{EL} with cyclic TBoxes and general concept inclusion axioms. Master thesis, TU Dresden, Germany, 2005.
- [SWR04] A Semantic Web Rule Language Combining OWL and RuleML, 2004. <http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/>.
- [TB07] A.-Y. Turhan and Y. Bong. Speeding up approximation with nicer concepts. In *Proc. of the 2007 Description Logic Workshop (DL 2007)*, 2007. Submitted to the Description Logics workshop.
- [TBK⁺06] A.-Y. Turhan, S. Bechhofer, A. Kaplunova, T. Liebig, M. Luther, R. Möller, O. Noppens, P. Patel-Schneider, B. Suntisrivaraporn, and T. Weithöner. DIG 2.0 – towards a flexible interface for description logic reasoners. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *In Proceedings of the second international workshop OWL: Experiences and Directions*, November 2006.
- [TH05] Dmitry Tsarkov and Ian Horrocks. Ordering heuristics for description logic reasoning. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*, pages 609–614, 2005.
- [The00] The Gene Ontology Consortium. Gene Ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.
- [TJ06] E. Torlak and D. Jackson. The Design of a Relational Engine. Technical Report MIT-CSAIL-TR-2006-068, MIT CSAIL, 2006.
- [TK04a] A.-Y. Turhan and C. Kissig. SONIC — Non-standard inferences go OILED. In D. Basin and M. Rusinowitch, editors, *Proc. of the 2nd Int. Joint Conf. on Automated Reasoning (IJCAR 2004)*, volume 3097 of *Lecture Notes in*

Computer Science, pages 321–325. Springer, 2004. SONIC is available from <http://www.tcs.inf.tu-dresden.de/~sonic/>.

- [TK04b] A.-Y. Turhan and C. Kissig. SONIC—System description. In V. Haarslev and R. Möller, editors, *Proc. of the 2004 Description Logic Workshop (DL 2004)*, number 104 in CEUR, 2004. See <http://CEUR-WS.org/Vol-104/>.
- [ton07] Techniques for ontology design and maintenance. Deliverable D13, TONES EU-IST STREP FP6-7603, January 2007.
- [Tur05] Anni-Yasmin Turhan. Pushing the SONIC border — SONIC 1.0. In Reinhold Letz, editor, *Proc. of Fifth International Workshop on First-Order Theorem Proving (FTP 2005)*. Technical Report University of Koblenz, 2005. <http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-13-2005.pdf>.