KRHyper Inside — Model Based Deduction in Applications

Peter Baumgartner
Ulrich Furbach
Margret Gross-Hardt
Thomas Kleemann
Christoph Wernhard

Institut für Informatik, Universität Koblenz-Landau, D-56070 Koblenz, Germany,
{peter,uli,margret,tomkl,wernhard}@uni-koblenz.de

Abstract. Three real world applications are depicted which all have in common, that their core component is a full first order theorem prover, based on the hyper tableau calculus. These applications concern information retrieval in electronic publishing, the integration of description logics with other knowledge representation techniques and XML query processing.

1 Introduction

Automated theorem proving is offering numerous tools and methods to be used in other areas of computer science. An extensive overview about the state of the art and its potential for applications is given in [9]. Very often there are special purpose reasoning procedures which are used to reason for different purposes, like e.g. knowledge representation [22] or logic programming [15].

The most popular methods used for practical applications are resolution-based procedures or model checking algorithms. In this paper we want to demonstrate, that there is an important potential for model based procedures. Model based theorem proving can be based very naturally on tableau calculi [19], and in particular there is a line of development, which started with the *SATCHMO* approach [28] and was later refined and extended towards the hyper tableau calculus [8].

In this paper three real world applications are depicted which all have in common, that their core component is a full first order theorem prover, based on the hyper tableau calculus. I.e. deduction is not used to produce or verify the software, but a deduction system is a part of the running application system. The three applications are

- information retrieval in electronic publishing
- reasoning in description logic and knowledge representation
- query answering and optimization in XML databases

These applications all stem from research and development projects, which are not dealing with automated reasoning primarily. The model generation theorem prover was an obvious tool for the purposes of these projects, because in each of the tasks it was not only a yes/no answer required, moreover was the model to be returned by the prover the answer to the query in the respective application.

In the following section we shortly depict the hyper tableau prover and on this basis we can describe the applications in the successive sections.

2 Theorem Proving with Hyper Tableau

2.1 Features

In the hyper tableau approach application tasks are specified using first order logic — plus possibly non-monotonic constructs — in clausal form. While a hyper tableau prover can be used straightforwardly to prove theorems, it also allows the following features, which are on one hand essential for knowledge based applications, but on the other hand usually not provided by first order theorem provers:

1. Queries which have the listing of predicate extensions as answer are supported.
2. Queries may also have the different extensions of predicates in alternative models as answer.
3. Large sets of uniformly structured input facts are handled efficiently.
4. Arithmetic evaluation is supported.
5. Non-monotonic negation is supported.
6. The reasoning system can output proofs of derived facts.
7. The system can be used as reasoner for a description logic, enhanced with rules and ABox reasoning capability.

Hyper tableau is a “bottom-up” method, which means that it generates instances of rule[1] heads from facts that have been input or previously derived. Derived facts are stored as lemmas. This has the heuristic effect of avoiding redundant re-computations and supports the use of the calculus for model generation, since it makes it possible to detect when a fixed point of rule application is reached.

If a hyper tableau derivation terminates without having found a proof, the derived facts form a representation of a model of the input clauses.[2]

[1] We use Prolog notation for clauses throughout this paper: A clause is viewed as rule “*Head* `:-` *Body*.”, where *Head* consists of its positive literals, combined by “`;`” (*or*), and *Body* consists of its negative literals, combined by “,” (*and*). If a clause contains only positive literals, i.e. is a fact or a disjunction, it is notated as “*Head*.”, if it contains only negative literals, as “`false :-` *Body*.”.

[2] The Herbrand model output consists of all ground instances of the derived facts. Since the derived facts must not necessarily be ground, in some cases they can characterize an infinite model.

A rule head may be a disjunction. In hyper tableau, disjunctions are handled by exploring the alternative branches in a systematic way. Backtracking can be used to generate one model after the other.

Of the features listed above, items 1 and and 2, the generation of answer sets, are made possible through model generation.

Large sets of uniformly structured input facts play a role comparable to base relations of databases in conventional applications, however nested and incomplete data structures can be represented by terms. So item 3 benefits from implementation techniques used in database systems, which can smoothly be integrated with the hyper tableau method.

The handling of special language constructs, items 4 and 5, is facilitated by two aspects of the controlled way in which the hyper tableau calculus builds up data structures: First, it does not generate new clauses with negative literals, which means that only input clauses have negative literals, and so information about the context in which special predicates will be evaluated is statically available at preprocessing. Second, the implementation of nonmonotonic operations such as negation as failure is facilitated by the possibility to detect when a fixed point of inferencing is reached. Intuitively speaking, we then know, that it is not possible to infer certain information, and can use this knowledge positively.

Regarding item 6, many first order theorem provers can output the proofs of refutations, albeit often in an idiosyncratic syntax that makes it difficult to process them further. For model generation, it is additionally desirable, that derivations of the facts belonging to a model are available.

Item 7, the practical suitability as a processor for description logic extended by rules and ABox reasoning, is a consequence of the other features. It is described in more detail in section 4.

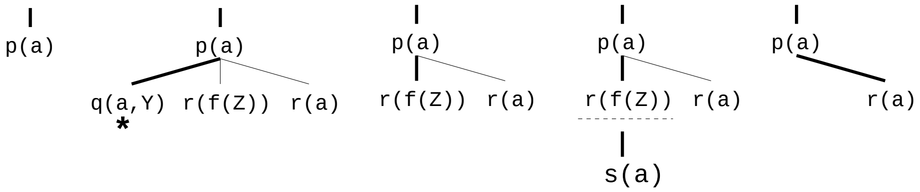
2.2 A Small Example

The following example illustrates how our hyper tableau calculus based system, *KRHyper*, proceeds to generate models. Figure 1 shows four subsequent stages of a derivation for the following input clauses:

```
p(a).                                       (1)
q(X, Y) ; r(f(Z)) ; r(X) :- p(X).           (2)
false :- q(X, X).                           (3)
s(X) :- p(X), not r(X).                     (4)
```

KRHyper provides stratified negation as failure. The set of input clauses is partitioned into *strata*, according to the predicates in their heads: if a clause c_1 has a head predicate appearing in the scope of the negation operator `not` in the body of c_2, then c_1 is in a lower stratum than c_2. In the example, we have two strata: the lower one containing clauses (1), (2) and (3), the higher one clause (4).

Stage (I) shows the data structure maintained by the method, also called *hyper tableau*, after the input fact (1) has been processed. One can view the



(I) (II) (III) (IV) (V)

Fig. 1. Stages of a KRHyper derivation

calculus as attempting to construct the representation of a model, the *active branch*, shown with bold lines in the figure. At step (I), this model fragment contradicts for example with clause (2): a model containing `p(a)` must also contain all instances of `q(a,Y)` or of `r(f(Z))` or `r(a)`. The model fragment is "repaired" by derivating consequences and attaching them to the hyper tableau: The corresponding instance of clause (2) is attached to the hyper tableau. Since it has a disjunctive head, the tableau splits into three branches. The first branch is inspected and proved contradictory with clause (3). This state is shown in (II).

Computation now tracks back and works on the second branch. With the clauses of the lower stratum, no further facts can be derived at this branch, which means, that a model for the stratum has been found, as shown in step (III). Computation then proceeds with the next higher stratum: `s(a)` can be derived by clause (4). Since no further facts can be derived, a model for the whole clause set has been found, represented by the facts on the active branch: {`p(a)`, `r(f(X))`, `s(a)`}, as shown in (IV).

If desired, the procedure can backtrack again and continue to find another model, as shown in state (V). Another backtracking step then finally leads to the result, that there is no further model.

2.3 The KRHyper System

Our system, *KRHyper*, implements the hyper tableau calculus by a combination of semi-naive rule evaluation [29] with backtracking over alternative disjuncts and iterative deepening over a term weight bound. It extends the language of first order logic by stratified negation as failure and built-ins for arithmetic.

For the applications described here, this system is used "embedded" in different ways: As knowledge maintenance and processing unit in the server component of a client-server system and as target system for the transformation of a higher level language, a knowledge representation language based on description logic.

3 Living Book — Electronic Publishing

Living Book [7] is an electronic book system based on the *Slicing Information Technology (SIT)* [13] for the management of personalized documents: Documents or textbooks are fragmented into small semantic units, so called *slices* or *units*, such as e.g. the definition of a concept, an example, an exercise or a paragraph. Slicing Information Technology evolved from an electronic library system for mathematics, the *ILF Mathematical Library*, which was developed within the *Deduction* research program of Deutsche Forschungsgemeinschaft in the nineties.

Meta data play an important role to describe dependencies among slices, which may originate from a single document or from different ones. Keywords can be assigned to slices to indicate their contents. The process of "slicing", i.e. fragmenting and annotating given documents such as manuals or mathematical textbooks, is partially automated, but usually needs some further manual work.

The Living Book system has a client-server architecture, which is shown in figure 2.

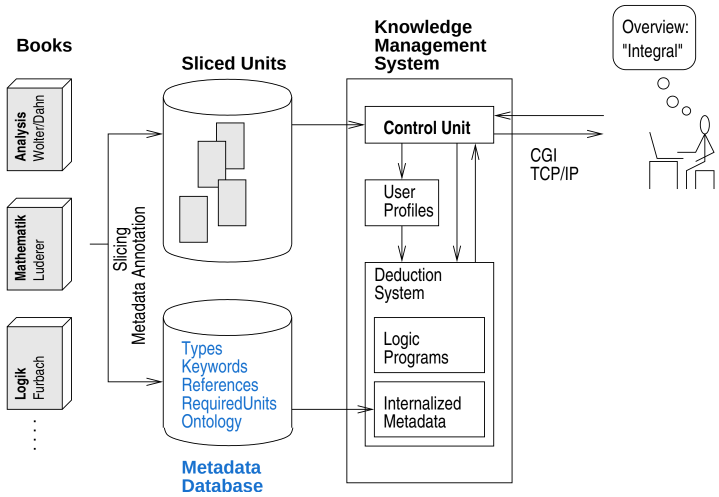


Fig. 2. Living Book — system architecture

3.1 The User View

A client side program, the *SIT-Reader*, offers all functionality of the system to the user through a common Web browser; figure 3 shows a screenshot.

To use the system, the user can mark units, like e.g. `analysis/3/1/15`[3] and `analysis/3/1/16` representing e.g. theorem 3.1.15 in the analysis book together with its proof. Then she can tell the system that she wants to read the marked units and gets a generated PDF document containing just those units. If the user thinks that this information is not sufficient for her understanding, she can tell the system to include all units which are prerequisites of the units selected.

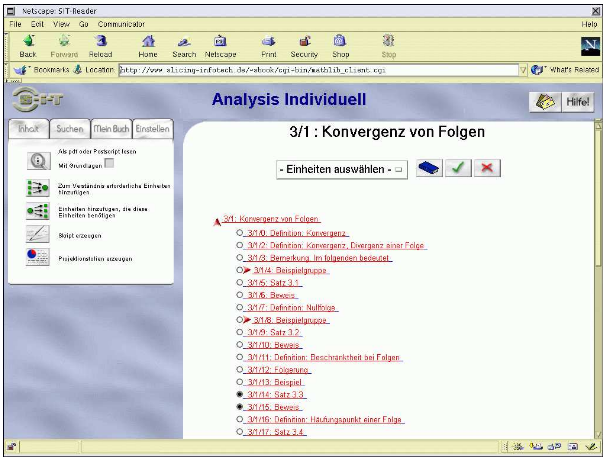


Fig. 3. Living Book — screenshot

But also more information about the user's profile can be incorporated in generating tailored documents: she may select a certain chapter, say e.g. chapter 3 containing everything about integrals in the analysis book. But instead of requesting all units from this chapter the user wants the system to take into account that she knows e.g. unit 3.1 already, and she possibly wants just the material that is important to prepare for an exam. Based on the units with their meta data, the deduction system can exploit this knowledge and combine the units to a new document (hopefully) fitting the needs of the user.

In conclusion, we not only have the text of the books, we have an entire knowledge base about the material, which can be used by the reader in order to generate personalized documents from the given books. If we are running the system with three books in combination, we have alltogether more the 12.000 facts and between 50 and 100 rules in the knowledge base.

3.2 The Knowledge Management System

From the viewpoint of deduction, the most interesting component of Living Book is the knowledge management system on the server side. As shown in figure 2, the knowledge management system handles meta data of various types: **Types** of units (*Definition, Theorem* etc), **Keywords** describing what the units are about (*Integral* etc), **References** between units (e.g. a *Theorem* unit about *Integral* refers to a *Definition* unit), and what units are **Required** by other units in order to make sense.

Further, a **User Profile** stores what is *known* and what is *unknown* to the user. It may heavily influence the computation of the assembly of the final docu-

[3] "/" is a binary function symbol, written as right-associative infix operator.

ment. The user profile is built from *explicit* declarations given by the user about units and/or topics that are known/unknown to him. This information is completed by deduction to figure out what other units must also be known/unknown according to the initial profile.

3.3 The Logic Behind

On a higher, research methodological level the deduction technique used in the knowledge management system is intended as a bridging-the-gap attempt. On one side, our approach builds on results from the area of *logic-based knowledge representation and logic programming* concerning the semantics of (disjunctive) logic programs (see [10] for an overview). On the other side, our KRHyper system used in Living Book is built on calculi and techniques developed for *classical first order classical reasoning*. To formalize our application domain we found features of both mentioned areas mandatory: the logic should be a first order logic, it should support a default negation principle, and it should be "disjunctive". To our knowledge, such a "cross-over" is novel, and therefore we will motivate the logic used by some examples now.

First order Specifications. In the field of knowledge representation, and in particular when non-monotonic reasoning is of interest, it is common practice to identify a clause with the set of its ground instances. Reasoning mechanisms often suppose that these sets are finite, so that essentially propositional logic results. Such a restriction should not be made in our case. Consider the following clauses, which are actual program code in the knowledge management system about user modeling:

```
unknown_unit(analysis/1/2/1).                   (1)
known_unit(analysis/1/2/_ALL_).                 (2)
refers(analysis/1/2/3, analysis/1/0/4).         (3)
known_unit(Book_B/Unit_B) :-                    (4)
        known_unit(Book_A/Unit_A),
        refers(Book_A/Unit_A, Book_B/Unit_B).
```

The fact (1) states that the unit named `analysis/1/2/1` is "unknown"; the fact (2), the `_ALL_` symbol stands for an anonymous, universally quantified variable. Due to the `/`-function symbol (and probably others) the Herbrand-Base is infinite. Certainly it is sufficient to take the set of ground instances of these facts up to a certain depth imposed by the books. However, having thus exponentially many facts, this option seems not really a viable one. The rule (4) expresses how to derive the know-status of unit from a known-status derived so far and using a refers-relation among units.

Default Negation. Consider the following program code, which is also about user modeling:[4]

[4] The `not` operator has been illustrated by the example in section 2.2.

```
%% Actual user knowledge:
known_unit(analysis/1/2/_ALL_).                  (1)
unknown_unit(analysis/1/2/1).                    (2)
refers(analysis/1/2/3, analysis/1/0/4).          (3)

%% Program rules:
known_unit_inferred(Book/Unit) :-                (5)
       known_unit(Book/Unit),
       not unknown_unit(Book/Unit).
unknown_unit_inferred(Book/Unit) :-              (6)
       unit(Book/Unit)
       not known_unit_inferred(Book/Unit).
```

The facts (1), (2) and (3) have been described above. It is the purpose of rule (5) to compute the known-status of a unit on a higher level, based on the `known_units` and `unknown_units`. The relation called `unknown_unit_inferred`, which is computed by rule (6) is the one exported by the user-model computation to the rest of the program.

Now, facts (1) and (2) together seem to indicate inconsistent information, as the unit `analysis/1/2/1` is both a `known_unit` and a `unknown_unit`. The rule (5), however, resolves this apparent "inconsistency". The pragmatically justified intuition behind is to be cautious in such cases: when in doubt, a unit shall belong to the `unknown_unit_inferred` relation. Also, if nothing has been said explicitly if a unit is a `known_unit` or an `unknown_unit`, it shall belong to the `unknown_unit_inferred` relation as well. Exactly this is achieved by using a default negation operation `not`, when used as written, and when equipping it with a suitable semantics[5].

Disjunctions and Integrity Constraints. Consider the following clause:

```
computed_unit(Book1/Unit1) ;
computed_unit(Book2/Unit2) :-
       definition(Book1/Unit1,Keyword),
       definition(Book2/Unit2,Keyword),
       not equal(Book1/Unit1, Book2/Unit2).
```

It states that if there is more than one definition unit of some `Keyword`, then (at least) one of them must be a "computed unit", one that will be included in the generated document (the symbol `;` means "or"). Beyond having proper disjunctions in the head, it is also possible to have rules without a head, which act as integrity constraints.

[5] Observe that with a classical interpretation of `not`, counterintuitive models exist. We use a variant of the perfect model semantics for stratified disjunctive logic programs.

4 Knowledge Representation Beyond Description Logic

In this section we will argue that automated deduction techniques, in particular those following the model computation paradigm, are very well suited for knowledge representation purposes. This is an argument leaving the mainstream of knowledge representation research, which currently has its focus on the development of description logic (DL) systems. We want to point out that we consider the DL direction of research extremely successful: it led to a deep insight into computational properties of decidable subclasses of first order reasoning; it made clear some interesting links to non-classical logics, and, moreover, DL systems are nowadays outperforming most modal logic theorem provers. Despite of these successful developments we find two reasons which motivate our approach to use a first order theorem prover for knowledge representation purposes instead of dedicated description logic systems.

First, even the key researchers in the field of description logics are stating some severe deficiencies of their systems (e.g. [18]): research into description logics focused on algorithms for investigating properties of the terminologies, and it is clear that for realistic applications the query language of description logic systems is not powerful enough. Only recently the community investigates seriously the extension of description logic systems towards ABox and query answering, which is not trivial [24, 23].

Second, the most advanced systems are essentially confined to classical semantics and do not offer language constructs for non-monotonic features (which are a core topic in another branch of the knowledge representation community). Although there are some results on extending DL languages with nonmonotonic features [4], it seems that this direction of research is vastly unexplored.

Additionally, it has been widely recognized that adding to the terminological language of DL a complementary knowledge representation scheme based on rules (as used in logic programs) would greatly improve expressivity. This issue is currently addressed in particular within the *Semantic Web* context, but no really convincing solution has been found so far [6, 16, 17].

According to the just said, our focus is on the development of a language and system that combines a terminological language with a rule like language and nonmonotonic features. The specifications may be "mixed", in the sense that concepts and roles defined in the terminological part may be used or further extended/constrained in the logic program part. Regarding computation with such specifications, we follow a model-computation paradigm. That is, a bottom-up procedure is employed that computes a (minimal) model of the whole specification. (The usefulness of the model-generation paradigm in general and in particular in conjunction with DL will be argued for below and in other parts of this paper.) The computation is using a naive transformation of the description logic syntax into first order predicate logic. Clearly, we are loosing decidability in the general case; however, being careful with the definition of knowledge bases in the application, we can retain decidability. With respect to performance we give some figures in the conclusion.

In the following we will roughly sketch the system we are targeting at.

4.1 Kernel + KRHyper

Our approach is oriented at the paradigm of logic programming and model-based theorem proving. Instead of starting with a small and efficient kernel language like ALC,[6] which is stepwisely extended towards applicability, we start with a rather general DL language and the rather general language of first order logic programs, and then we identify sub-languages that are decidable and practically feasible.

Our approach is a transformational one, which embeds DL into logic programming by translating DL constructs into logic programming constructs. This way, the semantics of the original specification is given the semantics of the resulting logic program. The largest subclass that we can handle is that of the Bernays-Schönfinkel fragment extended by a default-negation principle. The user of our system can decide to stay within this class or whether she wants to use some language constructs which leave this class.

Our approach can be summarized as

$$\mathsf{Kernel} + \mathsf{KRHyper}$$

where

- *Kernel* is an OIL-like language which is augmented by some additional constructs, like non-monotonic negation and second-order features (reification).
- *KRHyper* means the extended first order predicate logic which can be processed by our system. As a logic programming system, it provides rules, axioms, constraints and concrete domains.

4.2 The Kernel Language

OIL class definitions, e.g.

```
class-def defined carnivore
       subclass-of animal
       slot-constraint eats
           value-type animal
```

have a similar concrete syntax in our kernel language. Most parts of OIL are covered, in particular all kinds of class definitions, inverse roles, transitive roles etc. The constructs from the Kernel language are translated to our logic programming language following standard schemes.

Beyond this, we are able to handle the following points which are mentioned explicitly as missing in [18]:

[6] I.e. concept descriptions are formed using the constructors negation, conjunction, disjunction, value restriction and existential qualification.

Rules and Axioms. In addition to constructs in the syntax of the knowledge representation language we can use arbitrary formulae as constraints, rules or axioms. For instance, we can state in the rule part

```
dangerous(X) :- carnivore(X), larger_than(30,X).
```

to express sufficient conditions for being `dangerous`. The `larger_than` relation would be defined by the user as a binary predicate.

Using Instances in Class Definitions. Although it is well known (cf. [3]) that reasoning with domain instances certainly leads to EXPTIME-algorithms, it is very clear that exactly this is mandatory in practical applications. For instance, the previous example could also be supplied as[7]

```
dangerous <= carnivore & larger_than(30).
```

in the terminological part.

Default Reasoning. In our system we included a closed world assumption, such that we can use a default negation principle "\+ " following the perfect model semantics. Default negation may be used both in the rule part and in the terminological part. For the latter case, the previous example might more appropriately be written as

```
dangerous <= carnivore & \+ smaller_than(30).
```

Switching Back and Forth. One may switch back and forth between the terminological part and the rule part, by keeping in mind that concepts translate into unary predicates, and that roles translate into binary predicates.

ABoxes. Concrete instances of concepts (roles) are handled via unary (binary) predicates. This is a very natural and well-understood method for model generation procedures. For instance, from

```
dangerous <= carnivore & \+ smaller_than(30).
```

and the ABox consisting solely of

```
carnivore(leo).
```

the model generation prover will derive `dangerous(leo)`. Unlike as in other systems, no grounding in a preprocessing phase takes place, and the system is capable of computing with ABoxes consisting of tens of thousands of objects.

[7] Notice that description logic languages such as OIL usually permit concept definitions via equivalences (`<=>` in our syntax) or via necessary conditions (`=>` in our syntax). However, we start off with a concrete ABox that is assumed to implicitly represent a model of some TBox, and that can be extended to an explicitly represented model by using sufficient conditions, as shown.

Limited Second-Order Expressivity. Very often it is necessary to treat statements of the language as objects and to apply procedures for some kind of evaluation to them. This can be done in our context by meta-language constructs à la Prolog. For instance, via `concept_instance(Concept,Instance)` one has access to the `Concept` names where `Instance` is an instance of. For example,

```
all_dangerous(X) :-
       call(findall(Z,
           (dangerous(Y), concept_instance(Z,Y)),X)).
```

describes as a Prolog-list all the concepts that have an instance of the `dangerous` concept.

4.3 Sample Application

This method of using Kernel + KRHyper for reasoning in description logic is the core of an application we built for a major German bank. It is a knowledge management system, which is used as decision support for the communication department of the bank. An important characteristic of this system, is that it contains besides the TBox a large ABox, containing press articles and excerpts thereof. The reasoning mechanism is Kernel + KRHyper which in particular has to deal with the ABox. The system is extended with a graphical user interface, which allows easy modifications of the ABox and the TBox and which supports the usual queries one wants to get answered from knowledge base. The system is realized as a client-server architecture and can be used via an ordinary web browser. This knowledge management system is already in use in the bank and, currently it is extended versus automatic learning-based mechanisms, in order to extend the knowledge base.

Working on this application brought two rather trivial, but nevertheless important aspects to our attention. One is, that the user of such an application does not care at all, which technique is inside the system. From a deduction point of view, we are happy seeing a theorem prover running as part of an application software, the user, however, is only interested in things like reliability, efficiency and costs. We learned, that the use of an existing theorem prover helps to meet exactly these requirements. The second lesson we learned is, that it is extremely difficult for non-computer-scientists to use a language of description logic for defining concepts or to express complex queries. Although we defined a Windows-like graphical user interface, we found that users encounter difficulties to use it. Currently we are redesigning the interface of the system, such that the knowledge representation format is completely hidden behind functionalities which are only from the application domain.

5 Flexible Database Queries for XML Data

In the last years, semistructured data and in particular XML played an increasing role within the database community. Databases for XML data have evolved in the context of database integration tasks (*Enterprise Application Integration, EAI*); these databases have a strong focus on data that is structured irregularly, is implicit, incomplete and therefore often result in large schemas [1].

Various query languages for semistructured data have been proposed in order to deal with the complex and nested structure of these data. Many languages use path queries [26, 14, 12, 2, 25] which have emerged as an important class of browsing-style queries on the Web. Navigational database access by these path queries require the user to know lot of structural details. It has been pointed out that declarative query constructs are needed in order to reduce or even avoid explicit navigation through the data [20, 11].

Here, we present an approach that relies on the assumption that techniques from logics and in particular knowledge representation, as e.g. subsumption of XML types [27] may be useful for querying XML data.

5.1 Example

```
<university>
  <researcher>
    <name>Smith</name>
    <publications>
      <monograph>
        <title>Basics of databases</title>
        <author><name>Smith</name>
        </author>
        <isbn>7899</isbn>
        <subject>DB</subject>
      </monograph>
      <article>
        <title>Flexible queries</title>
        <author><name>Smith</name>
        </author>
        <author><name>Miller</name>
        </author>
        <proceeding>VLDB 2000
        </proceeding>
      </article>
      <publication>
        <title>XML Schema</title>
        <author>
          <name>Smith</name>
        </author>
      </publication>
    </publications>
  </researcher>
  <-- more researcher elements -->
  <library>
    <books>
      <book>
        <title>Databases</title>
        <author>
          <name>Smith</name>
        </author>
        <isbn>1234</isbn>
      </book>
      <book>
      <!-- more book elements -->
      </book>
    </books>
  <library>
</university>
```

Fig. 4. Example XML data

Let us consider an example XML document representing a university with a library and researchers working in the university. A library consists of books where each book has a title, an author and an isbn. Researchers have a name and an associated set of publications as e.g. articles, monographs or some general kind of publication without further specialization. Figure 4 shows an excerpt from the represented data.

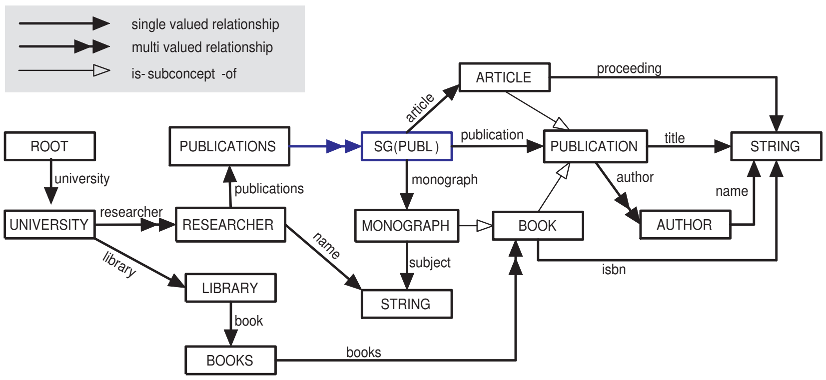


Fig. 5. Schema for database from figure 4

The content of a database is described by means of a schema; often for XML data *Document Type Definitions (DTDs)* are used. Recently, *XML Schema* [31] has evolved with the advantage, that user defined types can be represented; furthermore, besides referential relationships between elements of the XML data, XML Schema provides the possibility to model generalization and specialization relationships. Figure 5 uses a graphical notion to represent an example schema for the database in figure 4.

5.2 Querying the Data

Existing query languages use path queries, that navigate along the structures of the XML data. For instance, in order to access the name of all researchers of a university in *XPath* [30] we use `/university/researchers/researcher/name`. Path queries usually allow some form of "abbreviation". For instance, with `//researcher/name` we address all descendants of the "root" that are *researcher*-elements and navigate to their names. However, because path queries work directly on the XML data and not on the schema, it is not possible to query those elements from a data source, that belong to the same type or concept. In particular, in order to ask e.g. for all kinds of publications, we would have to construct the union of path queries navigating to publication, book, article and monograph, explicitly.

This problem has been addressed in [21], where the notions and the issues have been described. One possibility is to add a concept based query facility by means of graph based technology. Another possibility, presented here, is to rely on existing technologies and in particular systems from the area of description logics [5]. In DL, retrieving instances from a certain concept is a well known requirement and standard services are available out-of-the-box in existing DL systems such as *RACER* [22].

In order to use the DL-system, we represent the types in an XML schema by concept expressions in DL as follows. For every type c in the schema with attributes (outgoing edges) $a_1, a_2, \ldots, a_n$ leading to types $t_1, t_2, \ldots, t_n$ a concept expression

$$c \equiv \exists a_1.t_1 \sqcap \exists a_2.t_2 \sqcap \ldots \sqcap \exists a_n.t_n.$$

is introduced. Specialization edges from c to c' are represented by inclusion dependencies

$$c \sqsubseteq c'.$$

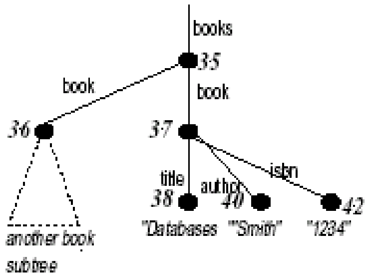


Fig. 6. Tree representation of XML documents

Furthermore, the data in an XML database is represented by ABox-facts. To this end, we represent each element in the database by a unique object identifier (see figure 6 for illustration). The appropriate ABox representation is sketched as follows. For every element o being an instance of concept c we introduce the fact

$$o : c$$

(e.g. o_{36}*:Book*) and furthermore, if the element o has a child o' with tag name a we write:

$$(o', o) : a.$$

For instance, (o_{35}, o_{37}) : *book*. This representation is used when using standard services from DL-systems. For instance, we now can use the service *retrieve-instances* applied to the concept PUBLICATION, in order to retrieve PUBLICATION, MONOGRAPH, BOOK and ARTICLE elements. Flexible query processing therefore makes use of TBox-reasoning (finding the relevant concepts and paths from the database description) and combines this with ABox reasoning and the retrieval of instances.

5.3 Path Completion

By *path completion* we mean, roughly, the problem of deriving from a concept name in a schema (the *start concept*) and an *end concept* a path through this schema connecting the start and the end concept. Such a path then can immediately be turned into a fully specified (e.g.) XPath query. Notice that this way the problems with XPath queries as explained in Section 5.2 can be avoided.

The question remains how to compute path completions. To this end, we propose to build on the DL formalization as described above. Then, it seems natural to appeal to some standard service offered by contemporary DL-systems and let it compute the path completion. Unfortunately we did not find a way of doing so. However, we succeeded in finding a solution based on *model computation*. It works as follows:

1. The start concept, say, c_s, is populated by filling the ABox with $a : c_s$, where a is a fresh name.
2. The DL specification of the schema graph is transformed into clause logic, using the well-known standard mapping. Also, the (singleton) ABox $a : c_s$ is transformed into a fact $c_s(a)$.
3. To the clause logic part the following two clauses are added, where `false :- c`$_e$`(X)` is the clause logic transformation of the concept $\neg c_e$:

   `end :- c`$_e$`(X).`
   `false :- not end.`

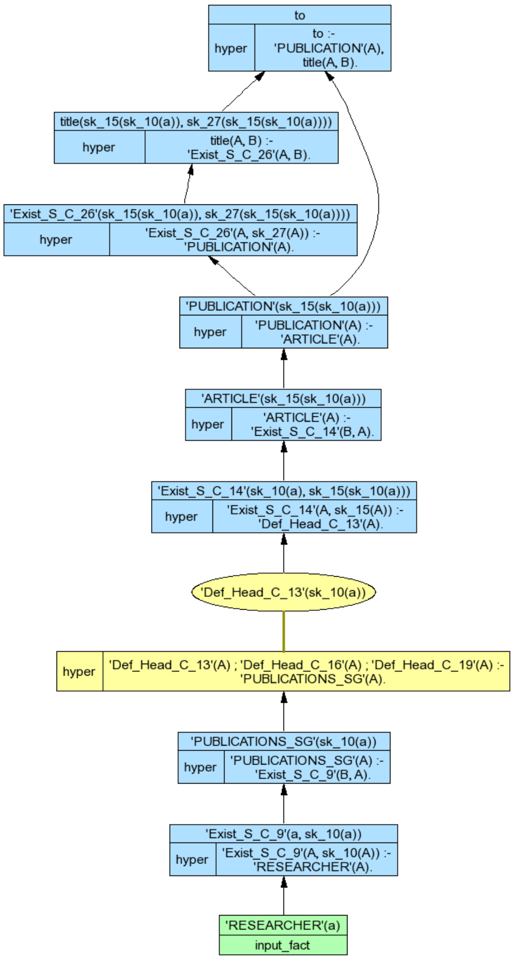


Fig. 7. Result of a KRHyper run. See text for explanation.

Now, speaking figuratively, computing a *minimal* model of the thus obtained clause set corresponds to labeling in the schema graph the start concept and propagate the label according to the concept hierarchy and via the roles encountered. Appealing to *minimal* model computation is an issue in order to avoid unwanted population of concepts. Notice that "substitution groups" [31] in the schema graph may introduce disjunctions in the head of the clauses, which may lead to different models or models that do not populate the end concept. It is exactly the purpose of the clauses in 3. above to avoid the latter problem. Observe that as small as the use of the default negation operator "`not`" may seem, it is a very useful feature to filter out unwanted models. The figure on the right shows the result of running our KRHyper prover, where the start concept is $\mathsf{RESEARCHER}$ and the end concept $\mathsf{PUBLICATION} \sqcap \exists \mathsf{title}.\top$.

6 Conclusion

The KRHyper theorem prover is an integral part of all three applications, that derive from quite different fields. Contrasting schemes that limit the use of a prover to the setup or configuration of an application, in our case the prover is a continuously used core component of the depicted systems. Any concerns about performance degradation imposed by the use of a full featured KRHyper system did not come true. Comparisons with a widely respected DL reasoner rendered KRHyper superior with respect to execution time as well as memory consumption. Based on these experiences we are continuing to investigate the applicability of deduction systems in upcoming projects.

References

1. S. Abiteboul. Querying semistructured data. In *ICDT*, pages 1–18, 1997.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
3. C. Areces, P. Blackburn, and M. Marx. A roadmap of the complexity of hybrid logics. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic*, number 1683 in LNCS, pages 307–321. Springer, 1999. Proc. of the 8th Annual Conf. of the EACSL, Madrid, September 1999.
4. F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proc. of the Third Int. Conf.*, pages 306–317, San Mateo, California, 1992. Morgan Kaufmann.
5. F. Baader and W. Nutt. Basic description logics. In F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors, *Description Logic Handbook*, pages 47–100. Cambridge University Press, 2002.
6. M. Balaban and A. Eyal. Dfl - a hybrid integration of descriptions and rules, using f-logic as an underlying semantics. In *DL*, 1996.
7. P. Baumgartner, U. Furbach, M. Gross-Hardt, and A. Sinner. `'Living Book' :- 'Deduction', 'Slicing', 'Interaction'.` – system description. In F. Baader, editor, *CADE-19 – The 19th Int. Conf. on Automated Deduction*, LNAI. Springer, 2003. To appear.
8. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In *Proc. JELIA 96*, number 1126 in LNAI. European Workshop on Logic in AI, Springer, 1996.
9. W. Bibel and P. H. Schmitt, editors. *Automated Deduction. A basis for applications.* Kluwer Academic Publishers, 1998.
10. G. Brewka, J. Dix, and K. Konolige. *Nonmonotonic Reasoning*, volume 73 of *Lecture Notes*. CSLI Publications, 1997.
11. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings of the Int. Conf. on Logic Programming (ICLP)*. Springer-Verlag LNCS, 2002.

12. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. ACM Symposium on Principles of Database Systems*, 1994.
13. I. Dahn. Slicing book technology - providing online support for textbooks. In H. Hoyer, editor, *Proc. of the 20th World Conference on Open and Distance Learning*, Düsseldorf/Germany, 2001.
14. J. V. den Bussche and G. Vossen. An extension of path expressions to simplify navigation in oject-oriented queries. In *Proc. of Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, 1993.
15. J. Dix, U. Furbach, and I. Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In A. Voronkov and A. Robinson, editors, *Handbook of Automated Reasoning*, pages 1121–1234. Elsevier-Science-Press, 2001.
16. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. Al-log: integrating datalog and description logics. *J. of Intelligent and Cooperative Information Systems*, 10:227–252, 1998.
17. F. M. Donini, D. Nardi, and R. Rosati. Description logics of minimal knowledge and negation as failure. *ACM Transactions on Computational Logic*, 3(2):177–225, 2002.
18. D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In *Proc. of the European Knowledge Acquisition Conf. (EKAW-2000)*, Lecture Notes In Artificial Intelligence. Springer-Verlag, 2000.
19. U. Furbach. Automated deduction. In W. Bibel and P. Schmitt, editors, *Automated Deduction. A Basis for Applications*, volume I: Foundations. Calculi and Refinements. Kluwer Academic Publishers, 1998.
20. L. L. G. Grahne. On the difference between navigating semistructured data and querying it. In *Workshop on Database Programming Languages*, 1999.
21. M. Gross-Hardt. Querying concepts — an approach to retrieve xml data by means of their data types. In *17. WLP - Workshop Logische Programmierung*, Technical Report. Technische Universität Dresden, 2002.
22. V. Haarslev and R. Möller. RACER system description. *Lecture Notes in Computer Science*, 2083:701, 2001.
23. I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic SHIQ. In D. MacAllester, editor, *Proc. of the 17th Int. Conf. on Automated Deduction (CADE-17)*, Germany, 2000. Springer Verlag.
24. I. Horrocks and S. Tessaris. A conjunctive query language for description logic aboxes. In *AAAI'2000, Proc. 17th (U.S.) National Conf. on Artificial Intelligence*, pages 399–404. AAAI Press/The MIT Press, 2000.
25. Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *PODS*, 2001.
26. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD*, 1992.
27. G. M. Kuper and J. Simen. Subsumption for XML types. In J. V. den Bussche and V. Vianu, editors, *Int. Conf. on Database Theory*. Springer LNCS 1973, 2001.
28. R. Manthey and F. Bry. SATCHMO: A theorem prover implemented in Prolog. In E. L. Lusk and R. A. Overbeek, editors, *Proc. of the 9th Conf. on Automated Deduction*, LNCS, pages 415–434. Springer, 1988.
29. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Maryland, 1989.
30. W3C. XPath specification. http://www.w3.org/TR/xpath, 1999.
31. W3C. XML Schema - part 0 to part 2. http://www.w3.org/TR/xmlschema-0, -1, -2, 2001.