Automated Deduction Techniques for the Management of Personalized Documents

Extended Abstract

Peter Baumgartner and Antje Blohm

Institut für Informatik
Universität Koblenz–Landau
D-56075 Koblenz
Germany

{peter,antje}@uni-koblenz.de

Abstract. This work is about a "real-world" application of automated deduction. The application is the management of documents (such as mathematical textbooks) that are decomposed ("sliced") into small units. A particular application task is to assemble a new document from such units in a selective way, based on the user's current interest.
It is argued that this task can be naturally expressed through logic, and that automated deduction technology can be exploited for solving it. More precisely, we rely on full first-order clausal logic (beyond Horn logic) with some default negation principle, and we propose a model computation theorem prover as a suitable deduction mechanism. On the theoretical side, we start with the hyper tableau calculus and modify it to suit the selected logic. On the practical side, only little modifications of an existing implementation were necessary.
Beyond solving the task at hand as such, it is hoped to contribute with this work to the quest for arguments in favor of automated deduction techniques in the "real world", in particular why they are possibly the best choice.

1 Introduction

This paper is about a "real-world" application of automated deduction. The application is the management of documents (such as mathematical textbooks) that are separated ("sliced") into small units. A particular application task is to assemble a new document from such units in a selective way, based on the user's interest.

The paper concentrates on describing the task to be solved and our attempt to formalize it with logic. Due to space reasons, and since it is not in the center of interest of this workshop, a technical exposition of the calculus is omitted here.

1.1 Tool and Project Context

Before describing how such a task can be formalized with logic and be solved with automated deduction techniques, it is helpful to briefly describe the tool context this work is embedded in.

This context is the *Slicing Books Technology (SBT)* tool for the management of personalized documents. With SBT, a document, say, a mathematics text book, is separated once as a preparatory step into a number of small units, such as definitions, theorems, proofs, and so on. The purpose of the *sliced* book then is to enable authors, teachers and students to produce personalized teaching or learning materials based on a selective assembly of units.

SBT is applied in the "real world": SBT was applied to a mathematics text book, which is explored commercially by the Springer Verlag, and SBT is the technical basis of the TRIAL-SOLUTION project[1]. The TRIAL-SOLUTION project aims to develop a technology for the generation of personalized teaching materials – notably in the field of mathematics – from existing documents (cf. `www.trial-solution.de` for more details).

Current work on SBT within the TRIAL-SOLUTION context is concerned with techniques to extend the capabilities by handling knowledge coming from the various sources. These sources include (i) different sliced books, (ii) a knowledge base of meta data on content (e.g. by keywords), didactic features, and interoperability interfacing, (iii) the user profile, including e.g. information about units known to him, and (iv) thesauri that help to categorize and connect knowledge across different books.

All these sources are to be taken into account when generating a personalized document. So far, no language was available to us to formulate in a "nice" way (convenient, friendly to change and adapt to new needs, efficient, ...) the computation of the personalized documents. Our approach was heavily motivated to come to a solution here.

1.2 Formalizing the Application Domain

In our approach, the document to be generated is computed by a model generating theorem prover. The computation is triggered by marking some unit U as a "selected unit". The background theory is essentially a specification of units to be included into the generated document. Such a specification is from now on called a *query*, and the task to generate the document from the query and the selected unit is referred to as *solving the query*. Here is a sample query:

(i) For each keyword K attached to the selected unit U, include in the generated document some unit D that is categorized as a definition of K; in case there is more than such unit, prefer one from book A to one from book B.
(ii) Include all the units that have at least one keyword in common with the keywords of U and that are of explanatory type (examples, figures, etc).

[1] TRIAL-SOLUTION is funded by the EU as part of its Information Society Technologies Programme (IST) within the EU's Fifth RTD Framework Programme.

(iii) Include all the units that are required by U.
(iv) Include U.

In our experiments we use sliced versions of two mathematics text books. Here are two sample units, one from each book:

```
    Ident: analysis/0/1/2/1
     Text: \item Mengentheoretische
           Grundbegriffe ...
     Book: Wolter/Dahn
     Type: Merksatz
   Refers: 0/1/0/2, 0/1/0/3, ...
 Requires:
     Keys: set, set intersection,
           set equality, ...
```

```
    Ident: gellrich/1/1/1/1/0
     Text: \definition{
           \textbf{Definition}:
           Eine \textbf{Menge} ...
     Book: Gellrich/Gellrich
     Type: Definition
   Refers:
 Requires:
     Keys: set
```

The *Ident* field contains a unique name of the unit in the form of a Unix file system subpath, matching the hierarchically organization of the units according to the books sectioning structure. The *Text* field contains the unit's text. The *Book* field contains the name of the book's authors. The *Type* field denotes the class the unit belongs to. The *Refers* and *Requires* fields contain dependencies from other units. The *Keys* field contains a set of keywords describing the contents of the unit.

Now suppose that the unit with Ident `analysis/0/1/2/1` is selected, and that the query from above is to be solved. The algorithm to actually compute a solution to the query conceptually proceeds by initially "marking" the selected unit, and then propagating marks according to meta-information like the refers-relation. This marking process is controlled by various parameters, e.g. a given bound on the maximum length of following the refers-relation, or the user model. Finally, from the thus marked units some units are filtered by e.g. given preferences or a given selection of desired types of units. The units surviving the filtering process constitute the solution.

First-Order Specifications. Some aspects in a logical formalization of the just outlined marker propagation scheme are straightforward, like the representation of a units meta-information and the representation of the selected unit identifier as a *fact* (as in `selected_unit(analysis/0/1/2/1)`). Beyond such facts, there are *rules*, which describe the marker propagation process. As a convenient language to formalize facts and rules we chose first-order Horn clause logic with default negation (also called *normal logic programs*, cf. [Llo87]).

In order to motivate the choice of this language, we display some sample facts and rules, which are taken from the *user-model.* Before doing so, some comments about the chosen concrete syntax might be appropriate. Basically, we are adopting Prolog-syntax: A *rule* is of the form H `:-` B_1`,` $\ldots$`,`B_n`.` $(n \geq 1)$, and a *fact* is of the form H`.`. A rule can be thought of to stand for a formula $\forall x_1 \cdots x_m \, (H \leftarrow B_1 \wedge \ldots \wedge B_n)$, where $x_1, \ldots, x_n$ are all the variables occurring

in $H \leftarrow B_1 \wedge \ldots \wedge B_n$ (facts are treated similarly). In Prolog-syntax, variables start with an upper-case letter or the symbol "_". Some function symbols, like "/", are written in infix-notation and may be declared to be e.g. left-associative.

```
/*
 * Facts: actual user knowledge:
 */
%% User says that he knows analysis/1/1, except analysis/1/1/2 :
known_unit(analysis/1/1/_).                                          (1)
unknown_unit(analysis/1/1/2/_).                                      (2)

/*
 * Lower layer: extend the known_topic and the known_unit relation:
 */
%% If a Topic is known, so are its subtopics:
known_topic(SubTopic) :-                                             (3)
        known_topic(Topic),
        subSuperConcept(SubTopic, Topic).

%% If a Unit is known, so are the refered units:
known_unit(ReferredBook/ReferredUnit) :-                             (4)
            known_unit(Book/Unit),
            references(Book/Unit, ReferredBook/ReferredUnit).

/*
 * Upper layer: combine known_topic and known_unit relations:
 */
unknown(Book/Unit, Keyword) :- unknown_unit(Book/Unit).              (5)

unknown(Book/Unit, Keyword) :- unknown_topic(Keyword).               (6)

unknown(Book/Unit, Keyword) :- not known_unit(Book/Unit),            (7)
                               not known_topic(Keyword).
```

Fig. 1. Some facts and a program fragment from the "user model".

Figure 1 depicts some facts and a program fragment from the "user model". The first fact (1) means that the user knows of the `analysis` book the section `1/1`. The variable `_` there represents all subsections. Similarly, fact (2) expresses that the whole subsection `1/1/2` is unknown.

As a design decision we fixed that `unknown_unit`-facts should override the information supplied by `known_unit`-facts. This is captured by the rule (5) in the "upper layer". Generally, the rules in the "upper layer" are used to derive

new facts of the form `unknown(Book/Unit, Keyword)`. These facts denote that `Book/Unit - Keyword` combinations are unknown, and they are explored by other parts of the program not shown here.

Beyond the `known_unit/unknown_unit` facts, there are `known_topic/unknown_topic`-facts (not depicted), which, together with rule (6), operate on keyphrases rather than unit specification, but otherwise work similarly.

The rules (3) and (4) define an extension of the base facts according to what is stated in the comments.

Finally, the rule (7) represents a default reasoning scheme. It expresses that by default a combination `Book/Unit - Keyword` is unknown, its `Book/Unit` or its `Keyword` is known.

Below in Section 2.3 this example is used to explain the working of our calculus. Some more comments can be made already here: due to the `/`-function symbol the Herbrand-base, and thus the set of ground instances of the program 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 is not really a viable one.

Some systems, like SATCHMO [MB88], do not rely on a preprocessing step that grounds the logic program. Instead, the program is required to satisfy the *range restriction* property, which means that every variable occurring in the head of a rule occurs also in the body of the rule. In essence, range restriction has the effect of grounding the rules during inference time in SATCHMO. Observe that the program above is not range-restricted due to facts (1) and (2) and rules (5) and (6). However, we feel that the formulation taken is a rather natural one. In conclusion, range restriction should not be enforced.

2 Automated Deduction

2.1 Nonmonotonic and Classical Logics

On a higher, research methodological level the work presented here is intended as a bridging-the-gap attempt: for many years, research in *logic-based knowledge representation and logic programming* has been emphasizing theoretical issues, and one of the best-studied questions concerns the semantics of default negation[2]. The problems turned out to be extraordinarily difficult, which warrant their study in detail. Fortunately, much is known today about different semantics for knowledge representation logics and logic programming. There is a good chance that a logic suitable for a particular application has been developed and studied in greatest detail.

Concerning research in *first-order classical reasoning*, the situation has been different. Not only theoretical issues, but also the design of theorem provers has traditionally received considerable attention. Much is known about efficient implementation techniques, and highly sophisticated implementations are around

[2] Like, for instance, Prolog's *Negation by finite failure* operator.

(e.g. SETHEO [GLMS94], SPASS [WAB+99]). Annual, international competitions are held to crown the "best" prover.

In essence, in this work we take as starting point a calculus originally developed for first-order classical reasoning – hyper tableaux [BFN96] – and modify it to compute models for a certain class of logic programs.

2.2 Logic Programming Semantics

We are using a sufficiently powerful logic, such that the queries can be stated in a comfortable way. In particular, including a default negation principle turned out to facilitate the formalization (cf. Section 1.2 above). As a restriction, we insist on stratified (normal) logic programs, which turns out not to be a problem[3].

We employ a model-theoretic semantics that is widely used in logic programming as the intended meaning of our programs[4]; solving the query means to compute such a model for the given program.

One point worth emphasizing is that we are thus *not* working in a classical theorem-proving setting (which is about proving consequences), and we are not employing classical first-order semantics; solving the query is, to our intuition, more naturally expressed as a model-generation task.

Fortunately, model computation for stratified programs is much easier than for non-stratified programs, both conceptually and in a complexity-theoretic sense. In particular, the two major semantics coincide, which are the stable model semantics [GL88] and the well-founded model semantics [VGRS91]. For propositional stratified normal programs, a polynomial time decision procedure for the model existence problem exists, which does not exist for the stable model semantics for non-stratified normal programs. Being confronted with large sets of data (stemming from tens of thousands of units) was the main motivation to strive for a tractable semantics.

2.3 The Calculus

One of the big challenges in both classical logic and nonmonotonic logics is to design calculi and efficient procedures to compute models for *first-order* specifications. Some attempts have been made for classical first-order logic, thereby specializing on decidable cases of first-order logic and/or clever attempts to discover loops in derivations of standard calculi (see e.g. [FL96,Pel99,Bau00,Sto99]).

In the fields of knowledge representation and in logic programming it is common practice to identify a clause set with the set of all possible ground

[3] A program is *stratified* if the call graph of the program does not have a cycle through a negative body literal. A simple example for a non-stratified program is $\{\mathtt{h} :- \mathtt{b},\ \mathtt{b} :- \mathrm{not}\,\mathtt{h}\}$, because h is defined recursively in terms of its own negation.

[4] To a ground program M is associated its minimal model $\mathcal{I}$ such that whenever $\mathcal{I} \models H$ then there is a clause $H :- B_1, \ldots, \mathrm{not}\, B_{m+1}, \ldots, \mathrm{not}\, B_n$ in M such that $\mathcal{I} \models B_1 \wedge \cdots \wedge B_m \wedge \neg B_{m+1} \wedge \cdots \wedge \neg B_n$. A non-ground program stands for the set of ground instances according to the underlying signature.

instances, both as a reference point to define the semantics and as input for inference mechanisms. It is assumed that these sets are finite, so that essentially propositional logic results. Of course, the "grounding" approach is feasible only in restricted cases, when reasoning can be guaranteedly restricted to a finite subset of the possibly infinite set of ground instances. Even the best systems following this approach, like the S-models system [NS96], quite often arrive at their limits when confronted with real data. Notable exceptions, i.e. reasoning mechanisms that directly operate on the first-order logic level, are described in [Bor96,ELS97,GMN+96,DS97].

In our application we are confronted with data sets coming from tens of thousands of units. Due to this mass, grounding of the programs before the computation starts seems not to be a viable option. Therefore, our calculus directly computes (representations of) models, starting from the given program, and without grounding it beforehand. In order to make this work for the case of programs with default negation, a novel technique for the representation of and reasoning with non-ground representations of interpretations is developed.

The calculus used here is obtained by combining features of two calculi readily developed – *hyper tableaux* [BFN96] and *FDPLL* [Bau00] – and some further adaptions for default negation reasoning. These two calculi were developed for *classical* first-order reasoning, and the new calculus can be seen to bring in "a little monotonicity" to hyper tableaux.

Let us briefly describe the main idea of the new hyper tableau calculus. The hyper tableau calculi are tableau calculi in the tradition of SATCHMO [MB88]. In essence, interpretations as candidates for models of the given clause set are generated one after another, and the search stops as soon as a model is found, or each candidate is provably not a model (refutation). Expressed a little more sloppily, new facts are derived from given facts, until a fixpoint is reached or a contradiction comes up. A distinguishing feature of the hyper tableau calculi [BFN96,Bau98] to SATCHMO and related procedures is the representation of interpretations at the first-order logic level. For instance, by taking in Figure 1 the fact (2) and instantiating the rule (5) appropriately, the new fact `unknown(analysis/1/1/2/_, Keyword)` is derived. Intentionally, this fact means that in the `analysis` book, all keywords attached to section 1/1/2/ and its subsections are unknown. Clearly, the representation on the first-order logic level is much more compact and efficient than the explicit representation that would consist of all respective ground instances.

The hyper tableau calculi developed so far do not allow for default negation. In the present work we therefore extend the calculus correspondingly. At the heart is a modified representation of interpretations. The central idea is to replace atoms – which stand for the set of all their ground instances – by pairs $A - \{E_1, \ldots, E_n\}$, where A is an atom as before, and E is a set of atoms ("Exceptions") that describes which ground instances of A are *excluded* from being true by virtue of A. For instance, from the fact (1) and the rule (7) from Figure 1 the calculus derives

$$\texttt{unknown}(\texttt{Book/Unit}, \texttt{Keyword}) - \{\texttt{unknown}(\texttt{analysis/1/1/\_}, \texttt{Keyword})\}$$

Intuitively, this means that in all books, all units are unknown, except Section 1/1 of the `analysis` book (which is declared to be known by the fact (1)).

We believe that the reasoning under a default negation principle based on a first-order logic representation is an interesting and rather underdeveloped topic in the logic programming literature. A detailed description of the calculus is beyond the scope of this paper and will be submitted elsewhere.

2.4 Other Approaches

In the previous sections, disjunctive logic programming was advocated as an appropriate formalism to model the task at hand. Undoubtedly, there are other candidate formalisms that seem well-suited, too. In the following we comment on these.

Prolog. Although the syntax chosen in our approach is the Prolog-syntax, our calculus works very different to the Prolog inference engine. Our calculus is a model generation approach, i.e. it works bottom-up, by deriving new facts from given facts. In contrast, Prolog works in a top-down manner, by reducing a given goal using the rules to the given facts.

Viewed from the technical side, one could use the Prolog `findall` built-in mechanism to simulate model computation. In order to make this work, some precautions would have to be taken, though. In particular, *explicit loop checks* would have to be programmed in order to let `findall` terminate. Because otherwise, for instance, alone the presence of a rule expressing transitivity causes `findall` not to terminate. On the other side, a built in loop check mechanism is a distinguishing and often mentioned advantage of virtually all bottom-up model generation procedures over Prolog.

Another disadvantage of Prolog is that the order of writing down the literals in rule bodies usually *is* significant as soon as negated literals are present. This is completely irrelevant in our approach.

In sum, we believe our approach relieves the programmer from issues to be considered in Prolog programming, thus makes programming *easier*.

SQL. It seems natural to view our application as a database application. Indeed, the meta data of the books considered in the TRIAL-SOLUTION project *are* stored in Postgres database. Therefore, one might consider using a database query language like SQL as an alternative to our logic oriented approach.

While this could certainly be done, we question the feasibility of using SQL. First, SQL does not allow for the definition of *arbitrary* recursive definitions. There is support for the special case of computing transitive closures of relations in newer versions of SQL. However, we felt in our programming that a more general device is needed. Second, in our approach it is very easy to incrementally compose a whole program by just throwing in some new rules and/or facts. We experienced that this feature is particularly useful in the early stages of program development, where quite some experiments are necessary to arrive at programs producing satisfactory results. This advantage would be lost when using a procedurally oriented language like SQL.

Description Logics. Description logics (DL) are a formalism for the representation of hierarchically structured knowledge about individuals and classes of individuals. Nowadays, numerous descendants of the original $\mathcal{ALC}$ formalism and calculus [SSS91, e.g.] with greatly enhanced expressive power exist, and efficient respective systems to reason about DL specifications have been developed [HST00].

From the sample query in Section 1.2 it can be observed that a good deal of the information represented there would be accessible to a DL formalization. The concrete units would form the so-called assertional part (A-Box), and general "is-a" or "has-a" knowledge would form the terminological part (T-Box). The T-Box would contain, for instance, the knowledge that a unit with type "example" *is-a* "explanatory unit", and also that a unit with type "figure" *is-a* "explanatory unit". Also, transitive relations like "requires" should be accessible to DL formalisms containing transitive roles.

In Section 1.2 it is argued that non-monotonic reasoning is suitable to model e.g. the user model. At the current state of our work, however, it is not yet clear to us if and how this would be accomplished using a DL formalism. Certainly, much more work has to be spent here. Presumably, one would arrive at a *combined* DL and disjunctive logic programming approach. This is left here as future work.

3 Putting Theory into Practice

A big concern of ours is the use of automated deduction in a "real world" application. This implies some aspects which are normally ignored by the developers of deduction systems. One aspect concerns technical issues, such as the need to enable the deduction system to cope with tens of thousands of facts (in typical theorem proving this does not come up). Another aspect concerns the problem that our intended users do not have the background knowledge to ask the "right" questions in a logic-oriented language.

3.1 User interface – or: How to Hide Logic from the User?

One important question in this context is the design of the interface between the automated deduction system and the user, i.e. the reader of the document. The user cannot be expected to be acquainted with logic programming or any other formal language (typically, this is what students should *learn* by reading the sliced books).

We provide two different strategies: predefined scenarios and piecewise combination of queries. A predefined scenario is a formalized description of units with respect to the situation of the reader. The fixing of the scenarios was incited by daily teaching problems. There are about 30 scenarios of the kind "Someone wants to know, which prerequisites are necessary to understand a topic.". The user only has to specify which topic he/she is interested in. The decision which types of units are relevant, whether referred and required units are also

integrated, and how the user's previous knowledge is taken into account are determined by the chosen scenario. A scenario can also be seen as a kind of macro for queries of the second strategy.

Since the predefined scenarios only provide a small subset of all possible queries we also offer the user a way to explicitly state in which kind of units he/she is interested. He/she describes the units in which he/she is interested in by selecting the attributes of her/his interest. The formalized query is automatically generated from the selected attributes.

3.2 No Deduction without Facts

In order to infer new knowledge some raw facts to start with are needed. In our application we deal with three different kind of data (i.e. facts): user data, data describing the relations between keywords, and data related to the units of a book.

Our user model holds data about the user's previous knowledge. The user has to specify which units and/or topics he/she knows and the prerequisite units/topics will be inferred. The preferences for certain books are also hold in the user model. The data hold in the model enable us to provide books adjusted to the user's needs.

In a thesauri, relations between keywords are represented. They normally contain the 'related-to'- and the 'is-subconcept-of'-relations . If the user asks for units concerning a certain topic (i.e. concept), it is possible to infer the subconcepts and/or the related concepts and to present him/her also the units dealing with the inferred concepts. With this approach we enable the user to get documents which cover the whole spectrum without stressing him/her with the knowledge about the relation of concepts.

The central part concerning the facts is the meta data (i.e. data about data) assigned to the units. Besides the type of the unit (definition, theorem, example etc.), the dependencies from other units are stored in the meta data part. Also the assignment of keywords to a unit are kept here. Without the relation between meta data and units it wouldn't only be impossible to present any sensible combination of the units, we would also lose the possibility to infer the user's previous knowledge.

4 Status of This Work and Perspectives

This work is not yet finished. Open ends concern in the first place some design decisions and practical experience with real data used by real students on a large scale.

Concerning design decisions, for instance, it is not quite clear what semantics suits our application best. It is clear that a supportedness principle (Section 2.2) is needed, but there is some room left for further strengthenings. Further experiments (i.e. the formulation of queries) will guide us here.

The calculus and its implementation are developed far enough, so that meaningful experiments are possible. The implementation is carried out in Eclipse Prolog. For faster access to base relations, i.e. the currently computed model candidate, the discrimination tree indexing package from the ACID term indexing library [Gra94] is coupled. Without indexing, even our moderately sized experiments seem not to be doable. With indexing, the response time for the sample query in Section 1.2 with a database stemming from about 100 units takes less than a second. Similar queries, applied to a repository of three books (yielding about 40000 facts) typically take less than 5 seconds to solve.

The question arises, if the techniques developed within this enterprise can be used in some other context. We plan to do so within one of the projects carried out in our group, the *In2Math* project. This project aims, roughly, to combine documents like sliced books with interactive systems like computer algebra systems and theorem provers. The projects targets at seriously using the resulting systems in undergraduate logic and mathematics courses. We are convinced that the techniques developed here can be used in the In2Math project. Beyond the obvious purpose to help students assemble personalized books, it is maybe possible to *teach* students participating in a logic course the logic programming techniques described here.

References

[Bau98] Peter Baumgartner. Hyper Tableaux — The Next Generation. In Harry de Swaart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 60–76. Springer, 1998.

[Bau00] Peter Baumgartner. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In David McAllester, editor, *CADE-17 – The 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 200–219. Springer, 2000.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper Tableaux. In *Proc. JELIA 96*, number 1126 in Lecture Notes in Artificial Intelligence. European Workshop on Logic in AI, Springer, 1996.

[Bor96] Sven-Erik Bornscheuer. Rational models of normal logic programs. In Steffen Hölldobler Günther Görz, editor, *KI-96: Advances in Artificial Intelligence*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 1–4. Springer Verlag, Berlin, Heidelberg, New-York, 1996.

[DS97] Jürgen Dix and Frieder Stolzenburg. Computation of non-ground disjunctive well-founded semantics with constraint logic programming. In Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski, editors, *Selected Papers of the Workshop on Non-Monotonic Extensions of Logic Programming in Conjunction with Joint International Conference and Symposium on Logic Programming 1996*, pages 202–226, Bad Honnef, 1997. Springer, Berlin, Heidelberg, New York. LNAI 1216.

[ELS97] Thomas Eiter, James Lu, and V. S. Subrahmanian. Computing Non-Ground Representations of Stable Models. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic*

Programming and Nonmonotonic Reasoning (LPNMR-97), number 1265 in Lecture Notes in Computer Science, pages 198–217. Springer-Verlag, 1997.

[FL96] Christian Fermüller and Alexander Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2):173–230, 1996.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of the 5th International Conference on Logic Programming, Seattle*, pages 1070–1080, 1988.

[GLMS94] Christoph Goller, Reinhold Letz, Klaus Mayr, and Johann Schumann. Setheo v3.2: Recent developments — system abstract —. In Alan Bundy, editor, *Automated Deduction — CADE 12*, LNAI 814, pages 778–782, Nancy, France, June 1994. Springer-Verlag.

[GMN+96] Georg Gottlob, Sherry Marcus, Anil Nerode, Gernot Salzer, and V. S. Subrahmanian. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science*, 166(1-2):221–262, 1996.

[Gra94] P. Graf. ACID User Manual - version 1.0. Technical Report MPI-I-94-DRAFT, Max-Planck-Institut, Saarbrücken, Germany, June 1994.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Symbolic Computation. Springer, second, extended edition, 1987.

[MB88] Rainer Manthey and François Bry. SATCHMO: a theorem prover implemented in Prolog. In Ewing Lusk and Ross Overbeek, editors, *Proceedings of the 9^{th} Conference on Automated Deduction, Argonne, Illinois, May 1988*, volume 310 of *Lecture Notes in Computer Science*, pages 415–434. Springer, 1988.

[NS96] Ilkka Niemelä and Patrik Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Bonn, Germany, 1996. The MITPress.

[Pel99] N. Peltier. Pruning the search space and extracting more models in tableaux. *Logic Journal of the IGPL*, 7(2):217–251, 1999.

[SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[SSW00] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for computing the well-founded semantics. *Journal of Logic Programming*, 2000. To Appear.

[Sto99] Frieder Stolzenburg. Loop-detection in hyper-tableaux by powerful model generation. *Journal of Universal Computer Science*, 1999. To appear in Special Issue on *Integration of Deduction Systems*. Guest editors: Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Peter H. Schmitt. Springer, Berlin, Heidelberg, New York.

[VGRS91] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.

[WAB+99] Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten Engel, Enno Keen, Christian Theobalt, and Dalibor Topić. System description: SPASS version 1.0.0. In Harald Ganzinger, editor, *CADE-16 – The 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 378–382, Trento, Italy, 1999. Springer.