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: `0/1/2/1`	Ident: `1/1/1/1/0`
Text: `\item Mengentheoretische Grundbegriffe ...`	Text: `\definition{ \textbf{Definition}: Eine \textbf{Menge} ...`
Book: `Wolter/Dahn`	Book: `Gellrich/Gellrich`
Type: `Merksatz`	Type: `Definition`
Refers: `0/1/0/2, 0/1/0/3, ...`	Refers:
Requires:	Requires:
Keys: `set, set intersection, set equality, ...`	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 `0/1/2/1` is selected, and that the query from above is to be solved. Some aspects in a logical formalization of this task are straightforward, like the representation of the units and the representation of the selected unit identifier as a fact (as in `selected_unit(0/1/2/1)`). The full formalization of the query proper is too long to be included here.[2] In order to motivate the whole approach, highlighting some parts of it might be helpful, though. Each of the following four parts demonstrates a different aspect of the formalization[3].

First-Order Specifications. In the field of knowledge representation 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:

```
wolter_dahn_unit(0 / _).            equal(X, X).
gellrich_gellrich_unit(1 / _).
```

The first two facts test if a given unit identifier denotes a unit from the respective book. 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

[2] It consists of 18 clauses.
[3] Written in a Prolog-style notation.

up to a certain depth imposed by the books. However, having thus exponentially many facts this option is not really a viable one.

Observe that *range restriction* (cf.[MB88]) does not apply to facts. The workaround, which is taken by some calculi, to enumerate the Herbrand-Base during proof search does not look too prospective either.

Some facts could conceivably be handled in such systems by building in the semantics of, e.g., `equal` into the prover. However, there remain cases where this seems not possible. A typical example is a clause like the following, which seems perfectly plausibly in our scenario:

```
knows_units(User,1 / 1 / _) :-
    current_user(User),
    learned(User,'set').
```

In sum, we have an "essential" non-ground specification.

Non-classical Negation. Consider the following clauses:

```
computed_unit(UnitId) :-
    candidate_def(UnitId,Key),
    not multiple_def(Key).
```

```
multiple_def(Key) :-
    candidate_def(UnitId1,Key),
    candidate_def(UnitId2,Key),
    not equal(UnitId1,UnitId2).
```

The `computed_unit` relation shall contain those units (named by unit identifiers `UnitId`) that go into the generated document. According to the left clause, this applies to any candidate definition unit for some keyword `Key` (derived by some other other clauses not described here), provided there is not more than one such candidate definition of `Key`. The second clause states the definition of the `multiple_def`-relation.

What is the intended semantics of `not`? *Classical* semantics is not appropriate, as it allows, for instance, arbitrary different terms to hold in the `equal`-relation. The correct intention, however, of `equal` is to mean syntactical equality. Likewise, classical semantics allows for counter-intuitive interpretations of `multiple_def`.

The *supported model* semantics (cf. Section 2.2), or possibly a stronger semantics (one that allows for more conclusions) seems to be intuitively correct. In brief, every atom true in a supported model must be justified by some clause, which means that the atom occurs in the head of some clause where the body is true in the model.

Disjunctions and Integrity Constraints. Consider the following two clauses:

```
computed_unit(UnitId1) ;
computed_unit(UnitId2) :-
    candidate_def(UnitId1,Key),
    candidate_def(UnitId2,Key),
    not equal(UnitId1, UnitId2).
```

```
false :-
        computed_unit(UnitId1),
        computed_unit(UnitId2),
        candidate_def(UnitId1,Key),
        candidate_def(UnitId2,Key),
        not equal(UnitId1,UnitId2).
```

The left clause states that if there is more than one candidate definition unit of some `Key`, then at least one of them must go into the generated document (the symbol `;` means "or"). The right clause states that not both of them must go into the generated document.

These clauses demonstrate that Horn clause logic is not sufficient. Any number of positive literals may appear in clauses.

2 Automated Deduction

When employing automated deduction techniques to solve a query like the one at the beginning of Section 1.2, some questions come up immediately: (a) what is an appropriate logic (syntax and semantics)? (b) How does a respective calculus look like? (c) is it efficient enough to solve real-world problems? (d) Why to do it at all with logic, and not, say, write a Prolog-program?

The most effort in our current project work is spent to answer questions (a) and (b). For space reasons, and since not really relevant for this workshop, the calculus is only sketched here. Although question (d) is addressed here, some more investigations are certainly necessary in this regard. In brief, our approach is completely declarative, and hence we believe that there are advantageous over more procedural oriented techniques.

Question (c) has to be answered by practice. At the moment, we have a prototypical implementation in Prolog, which, however, is coupled with a term indexing library for faster access to the base relations. In Section 4 a little more about our current experiences is reported.

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

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

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 and disjunctions in our logic turned out to facilitate the formalization (cf. Section 1.2 above). As a restriction, we insist on stratified specifications, which turns out not to be a problem. Stratified specifications are also called "logic programs" or simply "programs" in the sequel[5].

We define a special case of the *supported model* semantics [6] as the intended meaning of our programs, and solving the query means to compute such a model for the given program. This point is worth emphasizing: we are thus *not* working in a classical theorem-proving (i.e. refutational) setting; 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. There is little dispute about the intended meaning of stratified programs, at least for *normal* programs (i.e. programs without disjunctions in the head), and 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.

As mentioned above, we do not restrict ourselves to normal programs. Instead, we found it convenient to allow disjunctions in the head of clauses in order to express degrees of freedom for the assembly of the final documents. Also for the disjunctive case (i.e. non-normal programs), stable model semantics and well-founded model semantics have been defined (see e.g. [DFN01]). Both semantics agree to assign a *minimal model* semantics to disjunction. For instance, the program

$$A \vee B \leftarrow$$

[5] More precisely, the program clauses have the form $A_1 \vee \cdots \vee A_k \leftarrow B_1 \wedge \cdots \wedge B_m \wedge$ not $B_{m+1} \wedge \cdots \wedge$ not B_n, and 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 $\{A \leftarrow B,\ B \leftarrow \text{not}A\}$, because A is defined recursively in terms of its own negation.

[6] A model $\mathcal{I}$ of a (stratified and ground) clause set M is a *supported model* of M iff for every $A \in \mathcal{I}$ there is a clause $A \vee A_1 \vee \cdots \vee A_k \leftarrow B_1 \wedge \cdots \wedge B_m \wedge$ not $B_{m+1} \wedge \cdots \wedge$ not B_n in M such that $M \models B_1 \wedge \cdots \wedge B_m \wedge \neg B_{m+1} \wedge \cdots \wedge \neg B_n$.

admits two minimal models, which are $\{A\}$ and $\{B\}$. An equivalent characterization of minimal models is to insist that for each atom true in the intended model, there is a head of a true clause where *only* this atom is true. For our task, however, this preferred exclusive reading of disjunctions seems not necessarily appropriate. When assembling documents, redundancies (i.e. non-minimal models) should not be prohibited unless explicitly stated. In order not having to restrict to the minimal model semantics, we find the *possible model semantics* to be appropriate [Sak90]. With it, the above program admits all the obvious models $\{A\}$, $\{B\}$ and $\{A, B\}$. If the inclusive reading is to be avoided, one would have to add the integrity constraint

$$\leftarrow A \wedge B \ .$$

Unfortunately, the possible model semantics is costly to implement. At the current state of our developments, our calculus is *sound* wrt. the possible model semantics (i.e. any computed model is a possible model) but *complete* only in a weak sense (if a possible model exists at all, some possible model will be computed). In our current setting, solving the query means to compute *any* model of the program, so the lack of completeness is not really harmful. However, future experiments will have to show if this approach is really feasible.

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 field of logic programming, a common viewpoint is to identify a program with the set of all its ground instances and to apply propositional methods then. Notable exceptions are described [Bor96,ELS97,GMN$^+$96,DS97]. 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.

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 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 developed 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). A distinguishing feature of the hyper tableau calculi [BFN96,Bau98] to SATCHMO and related procedures is the representation of interpretations at the first-order level. For instance, given the very simple clause set consisting of the single clause

$$\mathrm{equal}(X, X) \leftarrow$$

the calculus stops after one step with the model described by the atom set $\{\mathrm{equal}(X, X)\}$, which stands for the model that assigns true to each ground instance of $\mathrm{equal}(X, X)$.

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, if the clause

$$\mathrm{different}(X, Y) \leftarrow \text{not } \mathrm{equal}(X, Y)$$

is added, then the search stops with the set

$$\{\mathrm{equal}(X, X) - \{\}, \mathrm{different}(X, Y) - \{\mathrm{different}(X, X)\}\} \ .$$

It represents the model where all instances of $\mathrm{equal}(X, X)$ are true, and all instances of $\mathrm{different}(X, Y)$ are true, except the reflexive ones.

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. Certainly, one could write a Prolog program to solve a query. When doing so, it seems natural to rely on the `findall` built-in to compute the extension of the `computed_unit` predicate, i.e. the solution to a query. Essentially, this means to enumerate and collect all solutions of the goal `computed_unit(U)` by means of the Prolog built-in backtracking mechanism. In order to make this work, some precautions have to be taken. 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 transitivity clause causes `findall` not to terminate.

It is obvious that a Prolog program along these lines would be much more complicated than the program in Section 1.2. On the other hand, our approach relieves the programmer from the burden of explicitly programming a loop mechanism, because it is built into the model computation procedure presented below. Indeed, this is a distinguishing and often mentioned advantage of virtually all bottom-up model generation procedures over Prolog.

XSB-Prolog. One of the view programming languages that works top-down (as Prolog) and that has built-in loop checking capabilities (as bottom-up model generation procedures) is *XSB-Prolog* [SSW00]. XSB-Prolog supports query answering wrt. the well-founded semantics for normal logic programs [VGRS91]. At the heart of XSB-Prolog is the so-called *tabling* device that stores solutions (instantiations) of goals as soon as computed. Based on tabling, it is even possible to compute extensions of predicates (such as `computed_unit`) and return them to the user.

The only problem with XSB-Prolog for our application is the restriction to *normal* programs, i.e. disjunctions in the head of program clauses are not allowed. Certainly, this problem could be circumvented by explicitly coding disjunctions in the program, but possibly at the cost of far less intuitive solution.

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 disjunctive and non-monotonic reasoning is suitable to model e.g. preference among units to be selected. 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.

[DFN01] Jürgen Dix, Ulrich Furbach, and Ilkka Niemelä. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In Andrei Voronkov and Alan Robinson, editors, *Handbook of Automated Reasoning*, pages 1121–1234. Elsevier-Science-Press, 2001.

[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.

[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.

[Sak90] C. Sakama. Possible Model Semantics for Disjunctive Databases. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proceedings First International Conference on Deductive and Object-Oriented Databases (DOOD-89)*, pages 337–351. Elsevier Science Publishers B.V. (North–Holland) Amsterdam, 1990.

[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.