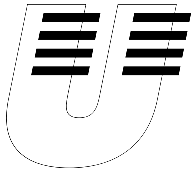


UNIVERSITÄT
KOBLENZ · LANDAU

Living Book –
Deduction, Slicing and Interaction

Peter Baumgartner, Margret Groß-Hardt,
Alex Sinner

2/2003

Fachberichte
INFORMATIK

Universität Koblenz-Landau
Institut für Informatik, Rheinau 1, D-56075 Koblenz

E-mail: researchreports@infko.uni-koblenz.de,
WWW: http://www.uni-koblenz.de/fb4/

Living Book – Deduction, Slicing and Interaction

Peter Baumgartner, Margret Gross-Hardt, Alex Sinner

Universität Koblenz-Landau
Institut für Informatik

{peter,margret,sinner}@uni-koblenz.de

Abstract

The Living Book is a system for the management of personalized and scenario specific teaching material. The main goal of the system is to support the active, explorative and self-determined learning in lectures, tutorials and self study. The Living Book includes a course on 'logic for computer scientists' with a uniform access to various tools like theorem provers and an interactive tableau editor. It is routinely used within teaching undergraduate courses at our university.

This paper describes the Living Book and the use of theorem proving technology as a core component in the knowledge management system (KMS) of the Living Book. The KMS provides a *scenario management* component where teachers may describe those parts of given documents that are relevant in order to achieve a certain learning goal. The task of the KMS is to assemble new documents from a database of elementary units called 'slices' (definitions, theorems, and so on) in a scenario-based way (like 'I want to prepare for an exam and need to learn about resolution').

The computation of such assemblies is carried out by a model-generating theorem prover for first-order logic with a default negation principle. Its input consists of meta data that describe the dependencies between different slices, and logic-programming style rules that describe the scenario-specific composition of slices. Additionally, a user model is taken into account that contains information about topics and slices that are known or unknown to a student. A model computed by the system for such input then directly specifies the document to be assembled.

This paper introduces the e-learning context we are faced with, motivates our choice of logic and presents the newly developed calculus used in the KMS.

1 Introduction

In the era of information technology, the need for new media concepts, like electronic books, has arisen. Despite the remarkable progress of electronic media, providing high quality and state of the art educational material is still quite a challenge. Printed books are still the backbone of current teaching technology, and printed books will certainly not disappear entirely in the near future. Their strengths rely on excellent graphic resolution and print quality as well as low prices and extreme portability. Furthermore, it is easy to browse a book to get a first impression on a subject as stated e.g. in [12].

However, shortcomings of printed books become obvious when focusing on search functionality or on explorative and creative work with learning material. Especially in (higher) education, where people with different background and learning habits want to assimilate specific knowledge, it is crucial to allow them to customize the content of their books according to their knowledge and style. Besides the requirement for *personalized* access to educational material from the learners perspective, there is also a strong motivation from the teacher's point of view in using the material for different purposes in different *scenarios*. A scenario describes those elements of learning resources that are necessary to achieve certain learning objectives. Usually, lectures are based on the same material but differ in focus and in depth of use of the various parts of the material. Depending on learning goals, learning groups etc. teachers may define different scenarios by describing the relevant parts of the underlying course material. These scenarios either may be used by a teacher for generating e.g. scenario specific sets of slides or even more important, these scenarios may be used by learners during preparation, e.g., for a certain exam.

Interactivity is another benefit which may be offered by electronic material (see e.g. [7]). In order to understand certain concepts, interactive systems (like e.g. computer algebra systems in mathematics or theorem provers in logics) enhance teaching resources and fit seamlessly into the relevant parts of the subjects. In particular, which interactive systems are offered in certain learning scenarios can be deduced directly from learning profiles of the users or from learning objectives defined by a teacher.

Living Book is an electronic book providing scenario specific and personalized access to one or multiple underlying documents and integrates interactive elements in order to support teachers and students with demonstrating examples and practical exercises. It includes a course on 'logic for computer scientists' with a uniform access to various tools like theorem provers and an interactive tableau editor. This course of the Living Book is routinely used within teaching undergraduate courses at our university.

In Living Book, a *scenario management* component represents the basis for

adapted delivery of learning resources wrt. to specific learning goals. A course, e.g., may be based on one or multiple textbooks; usually, a lecturer does not use the material completely or sequentially but selects those parts that fit the needs of the current learning situation. By means of scenario specification, relevant document parts and interactive tools for certain learning goals may be defined.

Scenario management, as realized in the Living Book system, has to our knowledge not been realized in other works so far. The salient feature of scenario management is twofold: firstly, a scenario describes which learning resources are needed in order to obtain certain learning goals and secondly, scenario management provides calculation mechanisms that adapt these resources on a user specific, personalized basis (e.g. existing user knowledge wrt. certain topics is taken into account). More concretely, our scenario management copes with knowledge about how scenarios are defined and combines it with user profiles. The latter refine knowledge about relevant parts in a document, but eventually also contain exceptions related to standard scenario definitions. The scenario management takes into account knowledge from three different sources: the different units of one or multiple underlying textbooks, scenario specifications describing relevant material for certain learning contexts and individual user profiles. Based on these knowledge sources, personalized documents specific to given scenarios are calculated on demand and delivered to the user.

In Living Book, scenario management has been implemented by means of a knowledge based system (see section 3) that relies on deduction techniques as provided in the KRHyper deduction system (see section 4). Based on the descriptions of different scenarios and on knowledge related to the users' learning profiles the deduction mechanism infers the relevant parts of the learning resources. The deduction component is an ideal means for combining various kinds of knowledge (about the underlying documents, scenarios and individual user learning goals) in order to derive adaptive personalized documents within the e-learning context. For the user, the deduction mechanism is transparent, that is, users benefits from the functionality without the need to understand processing details.

From an automated reasoning point of view, Living Book can also be seen as a vehicle to demonstrate the successful application of deduction techniques in real world applications. Scenario management is realized by means of model based deduction as implemented in KRHyper. We use a first order calculus with default negation. First order specifications provide the basis for efficient model generation and default negation allows for a much more compact specification than it would be possible without it, and it allows to conveniently express preferences. For instance, a scenario management may assume that a user who understands a certain unit of the document also understands certain presuppositions of this unit. Within a user profile, however, there may be an explicit indication (e.g.

logical fact), that one of the presupposed units is unknown. The knowledge in the user profile and in the specification from the scenario management therefore may result in inconsistencies. However, by means of default negation this inconsistency can be resolved; that is the knowledge from the user profile would be considered in this case as an exception to the scenario knowledge and has higher preference.

Actually, the e-learning application as described here represents a non-trivial application for deduction techniques wrt. the size of documents (resulting in many thousands of logical facts) and the number of inference rules to process. Our implementation shows that deduction techniques for document management applications like in e-learning is feasible. In fact, it becomes obvious from our approach, that the techniques presented in this paper may be applied to other document management applications, like e.g. the generation of problem-specific Unix man-pages or the assembly of personalized electronic newspapers, too.

The approach proposed in this paper is based on the following techniques:

- *Slicing Book Technology (SBT)* [11]
- *The Living Book Technology (LBT)* [7]
- *Automated Deduction Techniques for the management of personalized documents* [5]

The first two technologies are described briefly within the next section (section 2). SBT and LBT represent the technical equipment in order to make the Living Book a useful e-learning application; in particular, SBT separates given textbooks in a set of intra- and interrelated semantic units. LBT provides domain specific tools for students and teachers that can be used in lectures and exercises. Scenarios and their representation in the knowledge management system are introduced in section 3. The backbone of the Living Book, the deduction system implemented in KRHyper is described in section 4. Section 4 also presents the first order calculus that forms the formal basis of the deduction technique in KRHyper. In section 5 we summarize our approach and indicate directions for future work.

2 Slicing Book and Living Book — Background of Technologies

This paper is based on experiences gained within the project In2Math [26] supported by the German Ministry for Education and Research and the European Commission supported project Trial-Solution [37].

Both projects emphasize on the generation of personalized documents with a focus on interactive components, explorative use of teaching material and user modeling.

In2Math is a project headed by the University of Koblenz with members from 7 research groups and 5 partners from industry. Focus of the development in Koblenz is the *Living Book* system.

Living Book means personalized user oriented educational material together with interactive components. The Living Book aims at supporting the fundamental parts of undergraduate classes in theoretical computer science. The main goal is to support the active, explorative and self-determined learning in lectures, tutorials and self-study. Living Book relies on the Slicing Book Technology (see section 2.1) and a knowledge management system (section 3) for generating scenario specific and personalized teaching and learning material.

This section sketches the main ideas of the Slicing Book and gives an overview about the Living Book technology.

2.1 Slicing Book Technology

The Slicing Book technology takes documents or textbooks as input and splits them into small semantic units, so called *slices*, which may be e.g. a paragraph, a definition, or a problem in the original documents.

The splitting process takes formatting instructions and headlines into account in order to enrich the slices automatically with an initial amount of meta data. Additional meta data is then added semi-automatically. For instance, keywords have to be specified by the author(s) of a document. Meta data is associated with single slices and also with groups of slices; furthermore, relationships between slices are described by meta data. Partly this knowledge can be inferred from the document structure (e.g. by analyzing links that exist between different parts of a document); to some extent, though, semantical dependencies between slices are described manually (e.g. which slices of the document have to be understood by a learner in order to understand a certain subject). Relationships may not only exist between slices of one document but also between slices of different documents. The process of splitting documents and adding meta data is described in more detail in [12]

The slices resulting from the splitting process together with their meta data provide the basis for the personalized assembly of new documents[1]. For instance, a user can mark specific units (slices), like e.g. slice numbers `analysis/3/1/15` 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

[1] This is described in somewhat more detail in [4] and also in [5]. The present paper significantly extends these two papers.

marked unit and gets a PDF document containing just the units she wants to view. If the user thinks that this information is not sufficient to understand both the theorem and the proof, she can tell the system to include (automatically) all units which are prerequisites of the units selected.

Alternatively, a user 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. Based on the units with their meta data a reasoning mechanism can exploit this knowledge and combine the LaTeX based units to a new document perfectly fitting the needs of the user. The Slicing Book technology furthermore contains a glossary and keyword-based search that allows fast access to all units of interest. Hence, 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.

2.2 Living Book

Living Book aims at supporting the fundamental parts of undergraduate classes in theoretical computer science. The main goal is to support the active, explorative and self-determined learning in lectures, tutorials and self study [7]. Living Book uses the Slicing Book Technology in order to provide a personalization of content. Currently, a prototype of the Living Book system is used to teach *Logics for Computer Science* to undergraduate students at the University of Koblenz. Interactive components have been added to the book, allowing students to work on problems and solve exercises by means or in combination with interactive systems (e.g. theorem provers).

The Living Book relies on the idea of combining the advantages of printed books, e.g. excellent readability of mathematical and other scientific texts, with the flexibility and actuality of web based techniques. The statical sequential structure of classical textbooks is overcome by the possibility of a student to dynamically select those subjects she is interested in or she needs to know in order to prepare herself for e.g. an exam.

The main developments in Living Book are that (*i*) every unit may contain interactive elements referring to interactive domain tools developed within the In2Math project or that are available on the Internet and (*ii*) the integration of scenario management that provides different learning scenarios related to different learning objectives. Of course, these developments are not independent of each other but calculating scenario specific documents is done in combination with user profiles and the user's current status of knowledge related to subjects discussed in the textbooks as well as the status stemming from exercises done with interactive systems. Whereas interactivity will play a subordinate role within the

rest of the paper, scenario management will now be discussed in more detail.

2.3 Other Approaches

Interactive and personalized E-learning systems have been discussed in the literature.

In [31] an interactive electronic book (*i-book*) is presented. This i-book is devoted to teaching adaptive and neural systems for undergraduates in electrical engineering. The salient feature of this book is the tight integration of simulators demonstrating the various topics in adaptive systems and the incremental use of simulation during each chapter in order to develop successively on a certain subject. The i-book, though, does not cope with different learning scenarios or user profiles.

The paper [9] discusses perspectives for electronic books and emphasizes the need for personalized and user specific content. This article concentrates on personalized presentation of content, for instance by means of style sheet application to the content that is delivered to the user. Personalization applied to the content of the material as done in our approach via scenarios is not considered.

Based on an explicit representation of the structure of the concepts in the domain of interest and a user model [39] and [29] dynamically generates instructional courses. These approaches use planning techniques in order to determine the relevant materials on a per user basis. The user model in [39] describes the student's knowledge, and contains history information about previous sessions as well as personal traits and preferences. Interactivity is not integrated in the works described by [39]. In [29] an interactive and adaptive system is presented. Scenarios and user profiles are supported. Here, user profile distinguishes between *knowledge, comprehension* and *application* in order to reflect the different status of knowledge during learning.

These two approaches differ from Living Book in two main aspect: firstly, in Living Book, we have chosen a deduction based approach instead of planning techniques, and secondly, the user profile adapts according to what the users specify that they know. For instance, the user indicates those units that are already known. From this the system deduces everything that should be known, too, based on dependence relationships between knowledge units. In [39, 29], the user model is adapted based on information the system gathers from a user during a session, e.g. if a certain exercise has been successfully solved.

3 Scenario Management in Living Book

Besides individual selection of *interesting units* by learners, Living Book offers the possibility to select various scenarios defined by a teacher. A *scenario* can be

considered as a predefined view on the material defined with respect to certain learning objectives. The motivation is that lectures usually do not work with a given material from the beginning to the end, but instead select those parts of a book which are necessary to achieve the course objectives. In particular, different courses may have different objectives.

For instance, an introductory course on some subject has different requirements than an advanced course. The teacher may define course specific scenarios which result in different books and of course different sets of slides for each course. Another example may be that a teacher wants to specify how students should be preparing themselves for an exam. A scenario may be defined which covers those subjects which are relevant for the exam. A final example deals with a "workbench" scenario where students should learn to use different tools like theorem provers, tableaux, etc. for a certain subject like e.g. resolution. The 'logic for computer scientist' course in Living Book contains lots of exercises with embedded interactive systems. The "workbench" scenario may be used for a declarative description of those systems that are relevant for the resolution topic. Within this scenario, different formulae may be generated for different students (all formulae belong to the same kind of exercise but have e.g. different parameters) and then be processed by use of the appropriate systems.

Scenarios in the living book context can be considered as a kind of meta-template which defines what kind of information should be included in a personalized document.

In the rest of the section, we describe scenarios from the user's point of view, give an overview about the underlying knowledge management system architecture and motivate the logic based approach for representing scenarios and other knowledge sources in Living Book.

3.1 Using the Scenarios

A *scenario* describes a specific assembly task according to the reader's interest. A *student* scenario is a scenario that is potentially useful for students. As a typical example, a student might want to "get an overview about 'Integral' " (cf. the right of Figure 3, where the "student" communicates this scenario to the KMS).

Still, in order to apply a *Student Scenario*, we need to provide some more input. Figure 1 shows a screen-shot of the input mask of our system.

First, we need to specify *Topics* or *Units* with either keywords or unit designations. An example for a topic would be 'Integral', and an example for an unit would be `analysis/1/2`, which means "Section 1.2 of the book `analysis`"

Then we have to select a concrete *scenario*. Almost twenty scenarios are currently available. Figure 2 shows a list of currently implemented scenarios.

Finally, we can adjust a setting for the *Exploration of Related Topics*. This

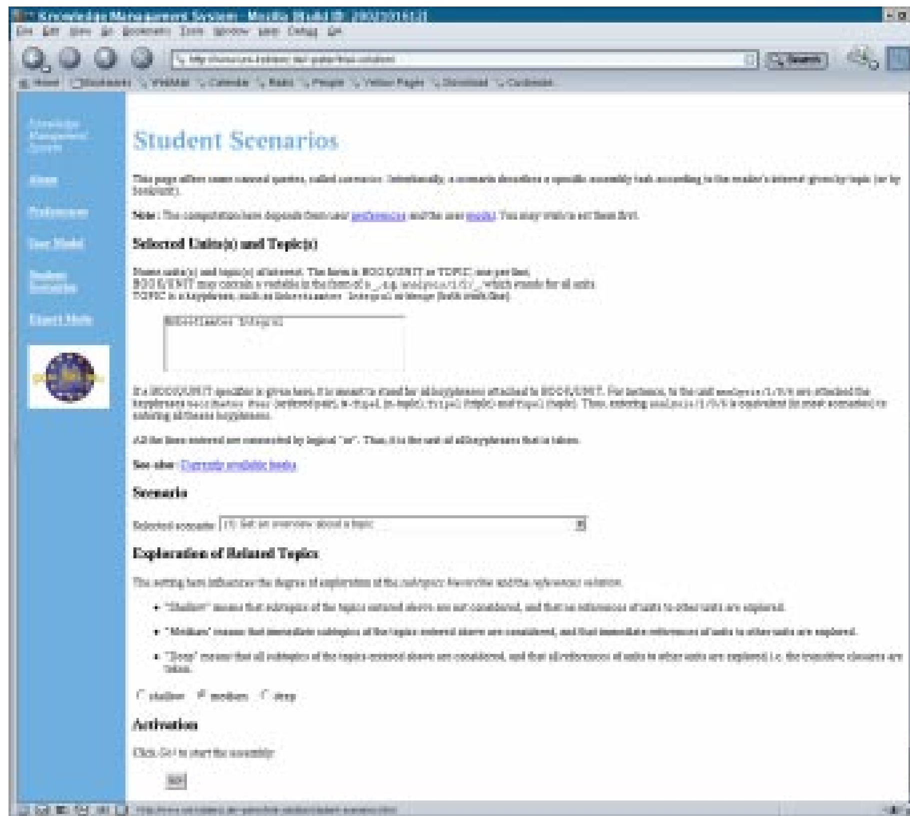


Figure 1: Test client: *Student Scenarios*

- Get an overview about a topic
- Get thoroughly into a topic
- Refresh knowledge of a topic
- Get deeper into a topic
- Get material to understand a topic
- Get prerequisites necessary to understand a topic
- Get material to understand the importance of a topic and what it is used for
- Get examples for a topic
- Get material to prepare for an oral exam on a topic
- Get material to prepare for a written exam on a topic
- Get exercises on a topic
- Get guidance for solving exercises on a topic
- Get all theory needed to solve exercises on a topic
- Get solutions to exercises on a topic
- Get exercises of the same kind
- Assemble a memory card / personal formulary
- Get references to literature / further reading

Figure 2: A list of *Student Scenarios*

setting, which is displayed at the bottom of Figure 1, influences the degree of exploration of the *subtopics hierarchy* in the ontology and the *references relation*. We can choose between *shallow*, *medium* and *deep*.

- *Shallow* means that subtopics of the topics entered are not considered, and that no references of units to other units are explored.
- *Medium* means that immediate subtopics of the topics entered above are considered, and that immediate references of units to other units are explored.
- *Deep* means that all subtopics of the topics entered above are considered, and that all references of units to other units are explored, i.e. the transitive closures are taken.

Besides this information (i.e. topics/units, concrete scenario, exploration flag), there is yet more information taken into account for the computation of the assembly, namely the *User Profile*. All units with which the user is already sufficiently familiar and which are not explicitly selected will not be included in the final document.

Furthermore a user also has the possibility to specify which books he prefers over others if two semantically identical slices are considered to be integrated into the personalized document. These preferences may however be exploited only for certain types of units. For example, if two definitions from different books are assigned the same key phrases, the same "thing" is defined. In contrast, if two theorems from different books are assigned the same key phrases, they usually are not the same. Hence, preferences for theorems should usually not be explored.

After all necessary information has been provided, the deduction system KRHyper is invoked. It takes as input the internalized form of the *User Data* module, and a logic program is selected according to the concretely chosen scenario (for every concrete scenario listed above under *Selected Unit(s) and Topic(s)* there is exactly one corresponding logic program). More information about the formalization of the domain and the logic programming techniques can be found below.

The result of the deduction system computation is returned to the *Control Unit*. It consists of a *description* of units that are to be assembled (i.e. a set of addresses like `analysis/1/2`).

Then, given this information, the text processors are invoked for the actual assembly of the units. Currently, there is one document for the computed units from each book where results were found. Finally, these documents are passed to the user.

Expert Mode. The purpose of the *Expert Mode* is best explained by an analogy: usually, Internet search engines offer at least two modes of operation: an easy-to-use mode, where one just types in some key phrase, and a refined search mode, where many parameters otherwise left implicit can be set. While the former mode corresponds in our case to the student scenarios, the latter mode corresponds to the expert mode. All parameters that influence the execution of the logic program can be set here.

The expert mode can be characterized as a "non-programmers way to program the KMS". Indeed, each of the *student scenarios* is nothing but a special setting of the parameters available in the expert mode.

In addition to the usual settings described above, we can specify a number of additional features, like what types of units (e.g. *All, Theorems, Definitions* etc.) are to be included in the assembly, or how far we follow references to/from our units etc.

The expert mode serves at least three purposes: first, it may be useful to the *end user* for the purpose of refined assembly. Second, it may be useful to *book authors* during the process of annotating their books with meta data, in order to explore the plausibility of the meta data in a very controlled way. Third, it may be an aid for *system programmers* when conceiving new scenarios.

3.2 KMS architecture

The KMS is designed as *client-server* architecture. When talking about the KMS in the sequel, we always mean the *server*, as it is the crucial part that implements the whole functionality.

The KMS consists mainly of three parts. It manages two databases, one containing sliced units and one containing the corresponding meta data. The main part of the KMS is however the *Knowledge Management Component.* It contains a *deduction system*, KRHyper, the user profiles, and a *Control Unit*, which manages the interactions between clients and the server. Figure 3 depicts the KMS schematically.

We store the following meta data in the KMS **Meta Data Database**:

- **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.

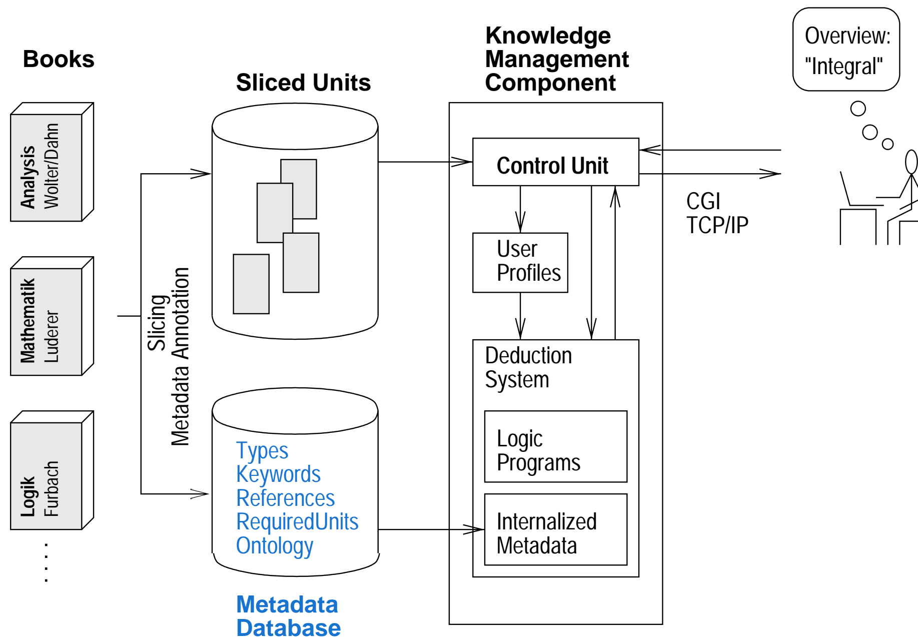


Figure 3: Knowledge Management System architecture

A **User Profile** stores mainly what is known and what is unknown to the user. It may heavily influence the computation of the assembly of the final document.

Currently 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 complemented through a deductive process which tries to figure out what other units must also be known/unknown according to the initial profile.

It should be noted that the current way of explicit user declarations can be easily substituted by other, more advanced techniques like automated user assessment.

3.3 A logic based approach

As said, the task of the Knowledge Management System (KMS) is to manage sliced books and their corresponding meta data and to compute the assemblies of new personalized documents from these slices.

The computation of such assemblies, or personalized documents, is a complex task. We tackle the problems with methods from the areas of logic programming and automated deduction. More specifically, the assembly tasks are specified

by *logic programs* that are processed by KRHyper [2], the automated deduction system at the core of the KMS. Here is some actual code from the KMS in order to get a flavor of the logic programs we use:

```
interesting_unit(Book/Unit) :-
    %% get a topic that was derived as interesting:
    interesting_topic(Topic),
    %% check that it can be derived that the topic is not known:
    not known_topic_inferred(Topic),
    %% we have to consider those units that are about the topic:
    unitKeyword(Book/Unit,Topic),
    %% make sure that the unit stems from a book enabled in preferences:
    book(Book),
    %% check that it can be derived that the unit is not known:
    not known_unit_inferred(Book/Unit).
```

This rule states a case of how to derive an "interesting unit". "interesting units" are at a middle layer in the computational hierarchy. If an "interesting unit" passes successfully further tests, it names a slice that goes into the final assembly. Our formalization is described in more detail below, and KRHyper is described in section 4.

In our opinion, a logic programming approach has several advantages over other, perhaps more traditional approaches like database (e.g. SQL), object-oriented or imperative programs. A discussion of this topic is deferred until Section 5.

3.3.1 Considerations on a Suitable Logic

On a higher, research methodological level the deduction technique used in the KMS is intended as a bridging-the-gap attempt. On the one side, our approach employs techniques and attempts to use results from the area of *logic-based knowledge representation and logic programming* (see e.g. [8] for an overview). One of the best-studied questions there concerns the semantics of non-monotonic knowledge representation languages like default logic, auto-epistemic logic, and logic programming languages with a default negation operator[3]. It turned out that the semantics of such logic involves several subtleties, which warrant their study in great detail. Fortunately, much is known today about different logics for knowledge representation purposes. There is a good chance that a logic suitable for a particular application has been developed and studied in greatest detail. However, research in this area has been emphasizing theoretical issues for many years.

[2] "KRHyper" stands for *K*nowledge *R*epresentation *Hyper* Tableaux.

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

On the practical side, most calculi and implemented systems for reasoning in such logics are essentially tied to the propositional level (see [13] for an overview).

On the other side, research in *first-order classical reasoning* has always had an additional focus. Not only theoretical issues, but also the design of theorem provers for first-order logic has traditionally received considerable attention. Much is known about efficient implementation techniques, and highly sophisticated implementations are around (e.g. SETHEO [20], SPASS [40]). Annual, international competitions are held to crown the "best" prover.

In our attempts 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". These properties are motivated now with examples from our application.

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 from the program code in the KMS about user modeling[4]:

```
unknown_unit(analysis/1/2/1).                                    (1)
known_unit(analysis/1/2/_ALL_).                                  (2)
known_unit(Book_B/Unit_B) :-                                     (3)
    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 (3) expresses how to derive the know-status of unit from a known-status derived so far and using a refers-relation among units.

The rules that we write down do not necessarily enjoy the *range restrictedness* condition (cf.[28])[5]. For instance, the first fact listed above is not range-restricted.

Many model-generation systems, like MGTP and Satchmo and its successors impose the range-restrictedness on the programs admissible for them. A

[4] We use Prolog notation.

[5] A program rule is *range-restricted* iff every variable occurring in the head of a rule also occurs in its body; facts are read as program rules without a body for the purpose of this definition.

workaround, which can be taken then, is to enumerate the Herbrand-Base during proof search. This means to consider all ground terms for the variables, which does not look too prospective.

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

Default Negation. Consider the following program code, which is also taken from the user modeling code from the KMS:

```
%% Actual user knowledge:
known_unit(analysis/1/2/_ALL_).                         (1)
unknown_unit(analysis/1/2/1).                           (2)

%% Program rules:
known_unit_inferred(Book/Unit) :-                       (3)
    known_unit(Book/Unit),
    not unknown_unit(Book/Unit).
unknown_unit_inferred(Book/Unit) :-                     (4)
    not known_unit_inferred(Book/Unit).
```

The facts (1) and (2) have been described above. It is the purpose of rule (3) to compute the known-status of a unit on a higher level, based on the `known_units` and `unknown_units`. The `unknown_unit_inferred` relation, which is computed by rule (4) 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 (3), 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[6].

Disjunctions and Integrity Constraints. Consider the following two clauses:

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

```
computed_unit(Book2/Unit2) :-
    computed_unit(Book1/Unit1),
    requires(Book1/Unit1, Book2/Unit2).
```

[6]Observe that with a classical interpretation of `not`, counterintuitive models of (1), (2) and (3) exist.

The left clause 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"). The right clause states that if a computed unit `requires` (the inclusion of) another unit, then that unit will be a computed unit, too.

A further discussion of this example is postponed until the section on "Semantics" below.

Beyond having proper disjunctions in the head, it is also possible to have rules without a head, which act as integrity constraints.

3.3.2 Semantics

As motivated above, our interest is in first-order specifications with a default negation operator. Quite some proposals in the logic programming community exist for assigning suitable semantics. As a prerequisite, the syntax of such specifications is restricted to clause logic, written in an implication-rule style, and the specifications are called programs then. The programs considered by the various semantics may differ, for instance, whether disjunctions in the head are allowed (see [8] for an overview). A widely studied class of programs is called *normal logic programs*, which consist of implications of the form

$$A \leftarrow B_1 \wedge \cdots \wedge B_k \wedge \mathsf{not}\, B_{k+1} \wedge \cdots \wedge \mathsf{not}\, B_n$$

where A and the B's are atoms and $n \geq k \geq 0$ and the symbol not is a distinguished symbol, intended as a default-negation operator.

A further important syntactical restriction is defined via the concept of *stratification*. Roughly, in stratified normal logic programs, no head atom may depend negatively from itself in the call graph underlying the program (as it would be the case e.g. in the program $A \leftarrow \mathsf{not}\, B$, $B \leftarrow A$). There is little dispute about the intended meaning of stratified normal programs, which is given by the perfect model semantics.

The two major semantics for possibly non-stratified normal logic programs are the stable model semantics [18] and the well-founded semantics [38]. For stratified programs they coincide with the perfect model semantics[7] [1]. Fortunately, computing the perfect model semantics of a propositional and finite normal program has low polynomial complexity (the same holds even for non-stratified programs for the well-founded semantics, but not for the stable model semantics). Our database consists of tens of thousands of units and leads to

[7] The perfect model semantics is defined for stratified programs only. It was developed before the stable model semantics and the well-founded semantics, and it can be considered as an important milestone at that time.

equally many facts. Being confronted with such large sets of data, the use of a tractable semantics, like the perfect model semantics, is mandatory.

The class of normal logic programs is generalized by the class of disjunctive logic programs, which consist of implications of the form

$$A_1 \vee \cdots A_m \leftarrow B_1 \wedge \cdots \wedge B_k \wedge \mathsf{not}\, B_{k+1} \wedge \cdots \wedge \mathsf{not}\, B_n$$

where $m \geq 0$. There exist various proposal for semantics for disjunctive logic programs, like the stable model semantics, the disjunctive well-founded semantics and the perfect model semantics for stratified programs (see again [8]).

Indeed, our interest is in disjunctive logic programs. Above, under "Disjunctions and Integrity Constraints" a motivating example from our application domain was given. Beyond the fact that it demonstrates the usefulness of disjunctive logic programs for us, the two clauses there can be used to argue for a central property of our KRHyper system, more precisely its underlying semantics: while the mentioned well-known semantics insist on a minimal-model property, the semantics that we use does not. To illustrate this point consider the clause set consisting of just the two propositional clauses $A \vee B$ and $B \leftarrow A$. All well-known semantics assign one single model to it, namely $\{B\}$. The model $\{A, B\}$ as well as all other models are rejected because they are not minimal. On the other side, it has been argued that models of the latter form might well "make sense" as well (see e.g. the possible model semantics [34]). Indeed, for our application, non-minimal models like $\{A, B\}$ make sense: suppose that a (ground) disjunction like computed_unit(analysis/4/2/0) $\vee$ computed_unit(analysis/5/4/3) has been derived by means of the left clause. Now, it might well be that the unit computed_unit(analysis/4/2/0) `require`s the unit computed_unit(analysis/5/4/3). A (non-minimal) model that assigns true to both computed_unit(analysis/4/2/0) and computed_unit(analysis/5/4/3), which would be computed by means of the right rule, perfectly matches the intuition of what should be computed. However, all the mentioned standard semantics would reject this model. Therefore, we will define a new semantics appropriate for our application. The models that we are interested in are called *weak perfect models*.[8] Each perfect model of a stratified disjunctive program will be a weak perfect model, but in general there are more weak perfect models for a disjunctive program than perfect models. For normal logic programs, the notions coincide.

The weak perfect model semantics will be introduced in the next section, together with a first-order calculus to compute weak perfect models.

[8]The possible model semantics [34] also allows non-minimal models. However, according to the intuition for our application, it would compute too many models. E.g., the program consisting of just $A \vee B$ has three possible models, which are $\{A\}$, $\{B\}$ and $\{A, B\}$. However, the last one will not be a weak perfect model. In fact, there is no "reason" why both A and B should be true in a model.

4 KRHYPER

The purpose of this section is to describe our deduction system KRHyper underlying the KMS. The motivation for the logic that can be treated by KRHyper was given in Section 3.3.1 above. Therefore, we will concentrate here on the techniques behind KRHyper. More precisely, we will describe the calculus behind the KRHyper deduction system.

4.1 Hyper Tableaux

The KRHyper calculus developed below is obtained by combining features of two calculi readily developed – *Hyper Tableaux* [6] and *FDPLL* [3] – and some further adaptations 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.

Before turning to a detailed account of the new calculus, we will review by way of example *on the propositional level* the calculus as defined in [6].

The *Hyper Tableau* calculus combines two things: the clustering of certain basic inference rules into a single one, much as in hyper-resolution [32], and the framework of clause normal form tableau (see [27]).

Instead of defining the Hyper Tableau calculus formally now as presented in [27] we will illustrate it with the following example.

Consider the following set of clauses, where clauses are given in implication form, such that $B_1 \vee \cdots \vee B_m \leftarrow A_1 \wedge \cdots \wedge A_n$ stands for the clause $B_1 \vee \cdots \vee B_m \vee \neg A_1 \vee \cdots \vee \neg A_n$.

$$A \leftarrow A \tag{1}$$

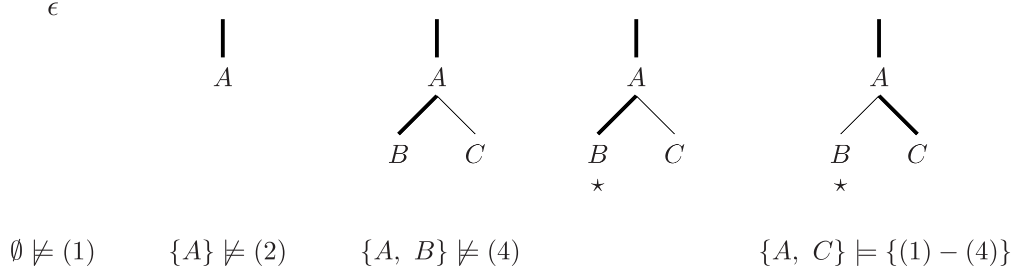
$$B \vee C \leftarrow A \tag{2}$$

$$A \vee D \leftarrow C \tag{3}$$

$$\leftarrow A \wedge B \tag{4}$$

In order to construct a hyper tableau for this clause set, we start with the empty tableau ϵ, which is given in the left part of Figure 4. We will discuss this derivation from left to right: If we consider clause (1), which we can understand as "in any model A has to hold", hence we extend our single (empty) branch of the tableau ϵ by the new leaf A. We arrive at a tableau with a branch which contains the (possibly) partial model A. Obviously clause (2) does not hold in this model, because (2) is stating "if in a model A holds, then B or C has to hold as well". Let us repair this, by extending our tableau by these two possibilities; we extend it by the disjunction $B \vee C$, which is expressed in the tableau by a new branching. The left branch $\{A, B\}$ of this new tableau, again is a (possibly)

partial model, but now we observe that there is a contradiction to clause (4), which is stating that A and B cannot be true together in any model; hence we know that this branch does not correspond to a partial model – we mark it as closed with an asterisk. The right branch of the tableau, however, although it could be further extended, is a model of the entire clause set (1)– (4).



$\emptyset \not\models (1)$ $\{A\} \not\models (2)$ $\{A,\ B\} \not\models (4)$ $\{A,\ C\} \models \{(1) - (4)\}$

Figure 4: A sample Hyper Tableau derivation.

We demonstrated the calculus only in the propositional case, but it can be extended to a complete and correct calculus for full first order clausal logic, and there are various improvements of its basic variant as introduced in [6].

Hyper tableau calculi are tableau calculi in the tradition of SATCHMO [28]. 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 [6, 2] to SATCHMO and related procedures is the representation of interpretations at the first-order level. For instance, given the clause set consisting of the single clause

$$\text{equal}(X, X) \quad \leftarrow$$

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

4.2 Formal Preliminaries

The usual notions of propositional and first-order logic are applied in a way consistent to [10]. A unary negation symbol not is assumed. It is used in front of atoms only, and $\mathsf{not}\, A$ is called a *default negative atom*. A *clause* is an expression of the form $A_1 \vee \cdots \vee A_m \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$, where $\mathcal{B}^+ = \{B_1, \ldots, B_k\}$ and $\mathcal{B}^- = \{\mathsf{not}\, B_{k+1}, \ldots, \mathsf{not}\, B_n\}$ are multisets of atoms and default negative atoms, respectively, where $m \geq 0$ and $n \geq k \geq 0$. Each atom $A_1, \ldots, A_m$ is called a *head atom*, each atom in $\mathcal{B}^+$ is called a *positive body atom* and each element in $\mathcal{B}^-$ is

called a *default negative body atom*. Clauses may also be written as

$$A_1 \vee \cdots \vee A_m \leftarrow B_1 \wedge \cdots \wedge B_k \wedge \mathsf{not}\, B_{k+1} \wedge \cdots \wedge \mathsf{not}\, B_n \ .$$

Clauses where $m = 1$ are called *normal clauses*, clauses where $m > 1$ are called *disjunctive clauses*, clauses where $m = 0$ are called *integrity constraints*, and clauses where $n = 0$ are called *facts*. A clause that is not a fact is also called a *rule*. A *program clause* is a normal clause or a disjunctive clause. A *program* is a finite set of clauses, and a *disjunctive program* contains at least one disjunctive clause.

Observe that there is no need for a "classical" negation sign $\neg$, because a disjunction like $A_1 \vee A_2 \vee \neg B_1 \vee \neg B_2$ can be written as $A_1 \vee A_2 \leftarrow B_1 \wedge B_2$.

When saying that a predicate symbol p *occurs* in a set of atoms or default negative atoms $\mathcal{B}$, we mean that p is the predicate symbol of some atom or default negative atom in $\mathcal{B}$. Now let C be a clause written as above. We say that a predicate symbol *occurs positively in C* iff it occurs in $\mathcal{A} \cup \mathcal{B}^+$, and it *occurs negatively in C* iff it occurs in $\mathcal{B}^-$. A predicate symbol *occurs in C* iff it occurs positively or negatively in C.

If $x_1,\ldots,x_n$ are variables and $t_1,\ldots,t_n$ are terms (over a given signature), we will denote by $\{x_1 \mapsto t_1,\ \ldots,\ x_n \mapsto t_n\}$ the substitution σ such that $x_i\sigma = t_i$ for all $i = 1,\ldots,n$ and $x\sigma = x$ for all other variables x. A substitution is a *renaming* iff it is a bijection of the variables onto themselves.

A *unifier* for a set Q of terms (or literals) is a substitution δ such that $Q\delta$ is a singleton. The notion of *most general unifier (MGU)* is used in the usual sense [10, e.g.], and a respective unification algorithm *unify* is assumed as given. The notation $\sigma = \mathit{unify}(Q)$ means that an MGU σ of Q exists and is computed by *unify* applied to Q. Failure (Non-unifiability) is noted as $\mathit{unify}(Q) = \mathrm{FAIL}$.

Quite frequently, a simultaneous unifier for some given unification problems $Q_1,\ldots,Q_n$ is to be computed, which is a substitution σ that is a unifier for every $Q_1,\ldots,Q_n$ individually. The notion of a most general unifier can be defined for the simultaneous case in the same way as above. It is well-known that a simultaneous most general unifier (simply called MGU as well) can be computed by iterative application of *unify* to $Q_1,\ldots,Q_n$ in sequence, thereby applying the most recently computed MGU to the remaining, still unprocessed unification problems. See [15] for a thorough treatment. Thus, we may suppose as given a simultaneous unification algorithm *s-unify* and write $\sigma = \textit{s-unify}(\{Q_1,\ldots,Q_n\})$ in analogy to $\sigma = \mathit{unify}(Q)$ above.

Now let A and B atoms. We say that A *is more general than* B, and write $A \gtrsim B$, iff there is a substitution σ such that $A\sigma = B$; A and B are called *variants*, written as $A \approx B$, iff $A \gtrsim B$ and $B \gtrsim A$ (equivalently: there is a renaming ρ such that $A\rho = B$); A is said to be *strictly more general* than B, and we write $A \gnsim B$, iff $A \gtrsim B$ but not $A \approx B$. B is also said to be a *strict*, or

proper instance of A then. If neither $A \gtrsim B$ nor $B \gtrsim A$ holds, then A and B are said to be *incomparable*.

Given an atom A and a set of atoms $\mathcal{A}$, we write $A \in_{\approx} \mathcal{A}$ iff $\mathcal{A}$ contains a variant of A.

Signatures are denoted by the letter Σ. All signatures considered are assumed to contain at least one constant symbol. The signature underlying a program $\mathcal{P}$ is denoted by $\Sigma_{\mathcal{P}}$.

By Σ-term we mean a term of signature Σ over a set of variables X. In the following, we will simply say "term" to mean a Σ-term. A term is *ground* iff it contains no variables. A *ground substitution for a term t* is a substitution γ such that $t\gamma$ is ground, and $t\gamma$ is called a *ground instance* of t then. These notions extend to atoms, literals and clauses in the obvious way. A *ground program* consists of ground clauses only. *Ground program of program P* is the ground program that is obtained by taking all ground instances of all clauses in P.

The *Herbrand universe* of a signature Σ is the set of all ground Σ-terms; it is denoted by $HU(\Sigma)$. Building on $HU(\Sigma)$, we denote by $HB(\Sigma)$ the Herbrand base of Σ, which is the set of all ground Σ-atoms. Finally, any subset of $HB(\Sigma)$ is called a *(Herbrand Σ-)interpretation $\mathcal{I}$*. As usual, the members of $\mathcal{I}$ are conceived as *true* ground atoms, while absent ground atoms are conceived as *false*.

Let $\mathcal{I}$ be an interpretation. A ground atom A and a ground clause $\mathcal{A} \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$ is evaluated wrt. $\mathcal{I}$ as expected, namely: $\mathcal{I}(A) = \mathit{true}$ iff $A \in \mathcal{I}$ and $\mathcal{I}(\mathcal{A} \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-) = \mathit{true}$ iff $\mathcal{B}^+ \subseteq \mathcal{I}$ and $\{B \mid \mathsf{not}\, B \in \mathcal{B}^-\} \cap \mathcal{I} = \{\}$ implies $\mathcal{A} \cap \mathcal{I} \neq \{\}$. Furthermore, for a non-ground clause C define $\mathcal{I}(C) = \mathit{true}$ iff $\mathcal{I}(C') = \mathit{true}$ for every ground instance C' of C.

We say that a finite set $\mathcal{B}$ of atoms or default negative atoms is *satisfiable in $\mathcal{I}$* iff there is a ground substitution γ for $\mathcal{B}$ such that $\mathcal{I}(\mathcal{B}\gamma) = \mathit{true}$, where the members of $\mathcal{B}$ are connected conjunctively.

As usual, $\mathcal{I} \models X$ means $\mathcal{I}(X) = \mathit{true}$ where X is a literal, a clause or a program (the clauses of which are connected conjunctively). If P is a program and $\mathcal{I} \models P$ holds, we also say that $\mathcal{I}$ is a *model* of P. Finally, a *minimal model of P* is a model of P such that no proper subset of it is also a model of P.

4.3 Weak Perfect Models

As mentioned near the end of Section 3.3.1, we are interested in stratified disjunctive programs that are interpreted under a variant of the perfect model semantics. We are now going to define all these notions precisely.

Definition 4.1 (Stratification)
A program P is *stratified* iff it is decomposable as $P = P_1 \cup \cdots \cup P_n$, for some $n \geq 0$, and such that the following holds, for $i = 1, \ldots, n$:

1. If a predicate symbol p occurs positively in a clause in P_i, then all clauses with p occurring in their head atoms are contained in $\bigcup_{j \leq i} P_j$.
2. If a predicate symbol p occurs negatively in a clause in P_i, then all clauses with p occurring in their head atoms are contained in $\bigcup_{j < i} P_j$.

□

We say that P *is stratified via* $P_1 \cup \cdots \cup P_n$ and call the P_i's the *strata of* P.

A trivial consequence of this definition is that any two rules in a stratified program with the same predicate symbol occurring their heads must be contained in the same stratum. This fact will be needed below.

Observe that the definition of stratification applies to all classes of programs: disjunctive, non-disjunctive, ground and non-ground programs. Furthermore, to a stratified non-ground program P that is stratified via $P_1 \cup \cdots \cup P_n$, a "natural" stratification can be assigned to its ground program P^{gr} by taking the stratification $P^{gr} = P_1^{gr} \cup \cdots \cup P_n^{gr}$, where P_i^{gr} is the ground program of P_i, for $i = 1, \ldots, n$.

Example 4.2 Consider the following program $\mathcal{P}$:[9]

$\mathcal{P}$:

$$
\begin{aligned}
P(x) \vee Q(x) &\leftarrow R(x) && (1)\\
P(x) &\leftarrow \mathsf{not}\, R(x) && (2)\\
P(x) &\leftarrow Q(x) && (3)\\
R(a) &\leftarrow && (4)
\end{aligned}
$$

The program $\mathcal{P}$ is stratified via $\{(4)\} \cup \{(1), (2), (3)\}$, and this is its only stratification. Examples for non-stratified programs are $\{P(x) \leftarrow \mathsf{not}\, P(x)\}$ and $\{P(x) \leftarrow \mathsf{not}\, Q(x),\ Q(x) \leftarrow \mathsf{not}\, P(x)\}$. □

The idea of stratification is to have the program partitioned in such a way so that decisions on default negative atoms in rule bodies are "permanent" in the sense that they are not affected by future decisions; in terms of strata, all decisions on default negative atoms occurring in a clause in stratum P_i must have been made during making the decisions on the atoms in $\bigcup_{j<i} P_j$. Observe that for every predicate symbol p, the collection of all clauses that contain p in the head must be found in *one single* stratum.

The intuition just explained is formalized in the following definition.

Definition 4.3 (Perfect Models)
Let P be a program, stratified via $P_1 \cup \cdots \cup P_n$, and $\mathcal{I}$ an interpretation. Define

$$\mathcal{I}_i := \{A \in \mathcal{I} \mid \text{the predicate symbol of } A \text{ occurs in a clause in } \textstyle\bigcup_{j \leq i} P_j\}\ .$$

[9] Here and below, the letters $P, Q, R, \ldots$ denote predicate symbols, $a, b, c, \ldots$ denote constants, $f, g, h, \ldots$ denote non-constant function symbols, and $x, y, z, \ldots$ denote variables.

Now, $\mathcal{I}$ is called a *perfect model of* P iff for all $i = 1, \ldots, n$, $\mathcal{I}_i$ is a minimal model of $\bigcup_{j \leq i} P_j$. □

Notice that this definition insists on the minimal model property of the submodels $\mathcal{I}_j$ for each collection of strata $\bigcup_{j \leq i} P_j$ *individually*. The minimal model of the last collection is (trivially) a minimal model of the whole program. As said, all those rules that contain a certain predicate symbol, say, p, must be located in the same stratum, say, P_j (for some $j \leq i$). From this fact and the mentioned minimal model property, it follows easily that the model $\mathcal{I}_j$ assigns *true* to *exactly* the same atoms with predicate symbol p as $\mathcal{I}$ does. Consequently, the truth values of the final model $\mathcal{I}$ can be determined stratum-wise, from lower to higher strata. It only needs to assign *true* to certain instances of head atoms of the rules in the current stratum, and leaving all other assignments untouched.

Further notice that any unsatisfiable program does not admit a single perfect model.

Example 4.4 The ground program $\{C \leftarrow C\}$ has the unique perfect model $\{\}$. Intuitively, deriving C by the rule would mean to support the conclusion C by assuming C at the same time; such a self-support is excluded.

Next, consider the following propositional program:

$\mathcal{P}$:

$$C \leftarrow C \quad (1)$$
$$B \leftarrow \mathsf{not}\, C \quad (2)$$
$$A_1 \vee A_2 \leftarrow B \wedge \mathsf{not}\, C \quad (3)$$
$$A_2 \leftarrow A_1 \quad (4)$$

The program $\mathcal{P}$ is stratified via $P_1 \cup P_2 = \{(1)\} \cup \{(2), (3), (4)\}$. It has exactly one perfect model, which is $\{B,\ A_2\}$. (Observe that $\{B,\ A_1,\ A_2\}$ is not a *minimal* model of P_2).

For a first-order logic example, consider the program $\mathcal{P}$ in Example 4.2 again. Assuming a signature that just contains the constant a,[10], $\mathcal{P}$ is equivalent to the following ground program:

$\mathcal{P}_a$:

$$P(a) \vee Q(a) \leftarrow R(a) \quad (1_a)$$
$$P(a) \leftarrow \mathsf{not}\, R(a) \quad (2_a)$$
$$P(a) \leftarrow Q(a) \quad (3_a)$$
$$R(a) \leftarrow \quad (4)$$

The program $\mathcal{P}$ has exactly one perfect model, $\{R(a), P(a)\}$, which can be easily read off from $\mathcal{P}_a$.

[10] More precisely: the signature must also include at least the predicate symbols used. However, here and below we feel this impreciseness in expression acceptable.

Now, if the signature would contain additionally the constant b, the following new ground instances become relevant:

$\mathcal{P}_b$:

$$P(b) \vee Q(b) \leftarrow R(b) \quad (1_b)$$
$$P(b) \leftarrow \mathsf{not}\, R(b) \quad (2_b)$$
$$P(b) \leftarrow Q(b) \quad (3_b)$$

With the extended signature, $\mathcal{P}$ still has one perfect model, which is $\{R(a), P(a), P(b)\}$. It is not difficult to derive it from $\mathcal{P}_a \cup \mathcal{P}_b$. □

Based on the definition of perfect models, the perfect model *semantics* of a program can be defined as the three-valued interpretation that assigns *true* (resp. *false*) to all ground atoms that are *true* (resp. *false*) in all perfect models of the program. All remaining ground atoms are undefined. However, the perfect model semantics as such is not in the center of our interest, while the computation of perfect models is.

More precisely, for a given program P our interest is to compute a class of models of P that actually may be a superset of the perfect models of P as defined above (the motivation for our deviation from the "standard" perfect models was given above in Section 3.3.2). Any such model will be called a *weak perfect model (of P)*. The weak perfect model of any (stratified) normal program P will coincide with its perfect model. Only for non-normal programs, i.e. for programs with disjunctions in the head of some rules, there will be a difference.

We start with a preliminary definition.

Definition 4.5 (Split Program)
Let P be a ground program. A *split program of P* is any ground non-disjunctive program P' such that P' can be obtained from P by replacing each disjunctive clause in P of the form $A_1 \vee \cdots \vee A_m \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$ by a normal clause $A_j \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$, for some j with $1 \leq j \leq m$.

By $\mathcal{S}(P)$ we denote the set of all split programs of P. □

Observe that a non-disjunctive program admits exactly one split program, which is the program itself.

Example 4.6 Consider again the ground program $\mathcal{P}_a$ from Example 4.4. There are two split programs, which are the following:

$\mathcal{P}_{a,P(a)}$:

$$P(a) \leftarrow R(a) \quad (1_a)$$
$$P(a) \leftarrow \mathsf{not}\, R(a) \quad (2_a)$$
$$P(a) \leftarrow Q(a) \quad (3_a)$$
$$R(a) \leftarrow \quad (4)$$

$\mathcal{P}_{a,Q(a)}$:

$$Q(a) \leftarrow R(a) \quad (1_a)$$
$$P(a) \leftarrow \mathsf{not}\, R(a) \quad (2_a)$$
$$P(a) \leftarrow Q(a) \quad (3_a)$$
$$R(a) \leftarrow \quad (4)$$

For the ground program $\mathcal{P}_b$, also from Example 4.4, there are also two split programs:

$\mathcal{P}_{b,P(b)}$:	$P(b) \leftarrow R(b)$	(1_b)	$\mathcal{P}_{b,Q(b)}$:	$Q(b) \leftarrow R(b)$	(1_b)
	$P(b) \leftarrow \mathsf{not}\, R(b)$	(2_b)		$P(b) \leftarrow \mathsf{not}\, R(b)$	(2_b)
	$P(b) \leftarrow Q(b)$	(3_b)		$P(b) \leftarrow Q(b)$	(3_b)
	$R(a) \leftarrow$	(4)		$R(a) \leftarrow$	(4)

In combination, the ground program $\mathcal{P}_a \cup \mathcal{P}_b$ has four (obvious) split programs; they are obtained by combining either $\mathcal{P}_{a,P(a)}$ or $\mathcal{P}_{a,Q(a)}$ with either $\mathcal{P}_{b,P(b)}$ or $\mathcal{P}_{b,Q(b)}$. □

As said above, the definition of stratification (Def. 4.1) applies to both disjunctive and non-disjunctive programs. Furthermore, to a stratified disjunctive program P that is stratified via $P_1 \cup \cdots \cup P_n$, a "natural" stratification can be assigned to any of its split programs P' by taking the stratification $P' = P'_1 \cup \cdots \cup P'_n$, where P'_i contains a normal clause $A_j \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$ if this clause replaces a disjunctive clause $A_1 \vee \cdots \vee A_m \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-$ in P_i.

In can be proven that $P'_1 \cup \cdots \cup P'_n$ is a partition of P', i.e. that no rule in P' occurs in more than one stratum. In essence, this holds because all rules in P that have a predicate symbol in common must be contained in the same stratum of P. So, when building a split program, if a normal clause replaces a disjunctive clause and this normal clause is already present, these two normal clauses must be in the same stratum. Based on this result, it follows easily from the definition of stratification that $P'_1 \cup \cdots \cup P'_n$ is indeed a stratification of P'.

The observation just made is important in view of the following definition.

Definition 4.7 (Weak Perfect Models)
Let P be a stratified program. An interpretation $\mathcal{I}$ is called a *weak perfect model of* P if it is a perfect model of some split program of the ground program of P. □

Expressed more operationally, the computation of a weak perfect model of a given program P consists of the following steps: first, the ground program P^{gr} of P is built. If P is stratified via $P_1 \cup \cdots \cup P_n$, it was mentioned above that P^{gr} can be stratified via $P_1^{gr} \cup \cdots \cup P_n^{gr}$. Then, some split program P' of P^{gr} is determined. It was argued above that a natural stratification $P'_1 \cup \cdots \cup P'_n$ of P' exists. Finally, the perfect model of P' is a weak perfect model of the given program P (if it exists).

Clearly, the steps of deriving a ground program and splitting of this program cannot be swapped. Even if the definition of split program would be prepared for non-ground programs, and even for restricted cases where head literals do not

share variables. To see this, consider the program consisting of the single fact $P(x) \vee Q(y) \leftarrow$ and suppose the signature contains just two constants a and b. Now, according to the definitions above, the ground program consists of the facts $P(a) \vee Q(a) \leftarrow$, $P(a) \vee Q(b) \leftarrow$, $P(b) \vee Q(a) \leftarrow$ and $P(b) \vee Q(b) \leftarrow$. It is easy to see that the set of weak perfect models of this ground program is a superset of the perfect models of the programs $P(x) \leftarrow$ and $Q(y) \leftarrow$.

Example 4.8 Consider again the propositional program $\mathcal{P}$ in Example 4.4 and its split programs mentioned in Example 4.6. The (non-disjunctive) program P_{A_1} there has the perfect model $\{B, A_1, A_2\}$, and the (non-disjunctive) program P_{A_2} there has the perfect model $\{B, A_2\}$. Hence, both models are weak perfect models of P. In contrast, as explained in Example 4.4, the single perfect model of P is $\{B, A_2\}$.

For the first-order program $\mathcal{P}$ in Example 4.2, its ground program (wrt. the signature that contains the constants a and b) was given in Example 4.4 as $\mathcal{P}_a \cup \mathcal{P}_b$. Recall from there that the unique perfect model of $\mathcal{P}_a \cup \mathcal{P}_b$, and hence of $\mathcal{P}$, is $\{R(a), P(a), P(b)\}$. The perfect models of the split programs of $\mathcal{P}_a \cup \mathcal{P}_b$ are the following:

Program	Perfect model
$\mathcal{P}_{a,P(a)} \cup \mathcal{P}_{b,P(b)}$	$\{R(a), P(a), P(b)\}$
$\mathcal{P}_{a,P(a)} \cup \mathcal{P}_{b,Q(b)}$	$\{R(a), P(a), P(b)\}$
$\mathcal{P}_{a,Q(a)} \cup \mathcal{P}_{b,P(b)}$	$\{R(a), P(a), Q(a), P(b)\}$
$\mathcal{P}_{a,Q(a)} \cup \mathcal{P}_{b,Q(b)}$	$\{R(a), P(a), Q(a), P(b)\}$

That is, $\mathcal{P}_a \cup \mathcal{P}_b$ has the two weak perfect models $\{R(a), P(a), P(b)\}$ and $\{R(a), P(a), Q(a), P(b)\}$, which are, by definition, the weak perfect models of the first-order program $\mathcal{P}$. □

4.4 Representation of Interpretations

Interpretations shall be represented in a more compact way than explicitly listing all the *true* ground atoms. This has two advantages: first, because the inference rules now reason on compactly represented sets in place of single atoms, *higher efficiency* can be achieved. And second, because in some cases infinite sets can be represented finitely, the calculus will terminate and compute results for more programs than those systems that perform ground-level reasoning.

The representation we use can be seen as an extension of the ARM concept (atomic representations of models, see [23]). With ARMs, an interpretation is represented as a set of non-ground atoms, which stands for the set of all its ground instances. Our representation extends this idea by the possibility to partially *exclude* ground instances represented by such non-ground atoms.

Definition 4.9 (Atom with Exceptions)
An *atom with exceptions*, or *AWE* for short, is a pair $A-\mathcal{E}$ consisting of an atom A and a finite set of atoms $\mathcal{E}$ such that $A \gtrsim E$, for every $E \in \mathcal{E}$. □

For simplicity of notation, we will simply write A instead of $A - \mathcal{E}$ if context allows.

Definition 4.10 (Cover, Most Specific Cover)
We say that an AWE $A - \mathcal{E}$ is a *cover of an atom* B iff

(i) $A \gtrsim B$, and

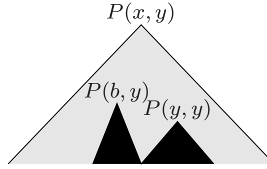
(ii) there is no atom $E \in \mathcal{E}$ such that $E \gtrsim B$.

A cover $A-\mathcal{E}$ of B is a *most specific cover (MSC) of* B *in* $\mathcal{N}$ iff there is no cover $A' - \mathcal{E}' \in \mathcal{N}$ such that $A \gtrsim A'$.

We say that a set of AWEs $\mathcal{N}$ *covers* B iff $\mathcal{N}$ contains an MSC of B in $\mathcal{N}$. □

In words, $A - \mathcal{E}$ covers B means just that B is an instance of A but is not an instance of any "exception to the instances of A" in $\mathcal{E}$.

Example 4.11 The following figure depicts in the solid region the atoms that $P(x, y) - \{P(b, y), p(y, y)\}$ covers.



Now consider a set of AWEs $\mathcal{N} = \{P(a, y)-\{P(a, a)\},\ P(x, y)-\{P(b, y), P(y, y)\}\}$. Both elements cover $P(a, c)$ in $\mathcal{N}$, but only the first element is a MSC of $P(a, c)$ in $\mathcal{N}$. Neither of them covers $P(a, a)$ in $\mathcal{N}$. Only the second element is a cover of $P(c, a)$ in $\mathcal{N}$, which is also an MSC of $P(c, a)$ in $\mathcal{N}$.

Suppose a signature that contains just the predicate symbol P and a constant a. Then, $\{P(x)-\{P(a)\}\}$ does not cover any *ground* atom. Now, if the signature would contain additionally a constant b, then the same AWE $\mathcal{N}$ would cover $P(b)$. □

As said, AWEs are intended to represent interpretations. The following definition makes this precise.

Definition 4.12 (Induced Interpretation)
Let Σ be a signature and $\mathcal{N}$ a set of AWEs. The *(Herbrand* Σ*-)interpretation induced by* $\mathcal{N}$, denoted by $\mathcal{I}_{\mathcal{N}}$, is the set of all ground Σ atoms that $\mathcal{N}$ covers. □

Observe that the interpretation induced by a set of AWEs $\mathcal{N}$ is indeed an interpretation as defined in Section 4.2. A ground atom A thus is *true* in it iff some AWE in $\mathcal{N}$ covers A. More precisely, A is assigned *true* iff it is an instance of A', for some $A' - \mathcal{E}$ in $\mathcal{N}$, and A is not an instance of any atom in $\mathcal{E}$.

The previous example 4.11 shows an important aspect of AWEs: without knowing the signature, it is impossible to say what ground terms it covers. More precisely, it is impossible in that case to *generate* from a given AWE all the ground instances it covers. However, there is a simple algorithm to determine if a given AWE covers a given ground atom. This algorithm extends in a trivial way to the case for finite sets of AWEs.

For reasons that will become clear later, it is important that the AWEs the calculus computes with satisfy a certain property. In terms of Example 4.11, the AWE $P(x) - \{P(a)\}$ will only be acceptable, if there is (at least) one ground atom different from $P(a)$. In general, an AWE is only acceptable, if it covers at least one ground atom. As just argued, this is impossible to say unless the underlying signature is taken into account.

Our approach to determine if an AWE is acceptable in the sense just mentioned is based on a normalization operation on AWEs (which takes the signature into account). From the normalized AWE one can easily read off if the given AWE is acceptable. We are going to describe this normalization operation now.

Definition 4.13 (Instance Set)
Let Σ be a signature and A an atom. The *instance set of* A, denoted by $IS_\Sigma(A)$, is the set of all Σ-ground instances of A. For a set of atoms $\mathcal{E}$, the *instance set of* $\mathcal{E}$, denoted by $IS_\Sigma(\mathcal{E})$, is the union of the instance sets of all atoms in $\mathcal{E}$. □

Instead of $IS_\Sigma(A)$ ($IS_\Sigma(\mathcal{E})$) we will simply write $IS(A)$ ($IS(\mathcal{E})$) if Σ is clear from the context.

Definition 4.14 (Normalization of Atom Sets)
Given a signature Σ and a finite set of atoms $\mathcal{E}$. A *normalization of* $\mathcal{E}$ is a set of atoms $\mathcal{E}^\nu$ satisfying the following conditions:

1. $IS(\mathcal{E}) = IS(\mathcal{E}^\nu)$.
2. For every non-empty subset $\mathcal{E}' \subseteq \mathcal{E}$ and all atoms A': if $IS(A') = IS(\mathcal{E}')$, then there is an atom $A \in_\approx \mathcal{E}^\nu$ such that $A \gtrsim A'$.
3. There are no two different atoms $A, A' \in \mathcal{E}^\nu$ such that $A \gtrsim A'$.

□

A naïve algorithm to compute a normalization of a given set of atoms $\mathcal{E}$ can be sketched as follows.

1. Chose a subset $\mathcal{E}' \subseteq \mathcal{E}$ and an atom A such that $A \gtrapprox A'$, for every $A' \in \mathcal{E}'$, and such that $IS(A) = IS(\mathcal{E}')$.

   (There are only finitely many atoms A such that $A \gtrapprox A'$, for each $A' \in \mathcal{E}'$, modulo renaming. The test for $IS(A) = IS(\mathcal{E}')$ can be carried out based on matching A against certain, finitely many Σ-atoms, the term depth of which is bounded by the term depth of the atoms in $\mathcal{E}'$. This together guarantees the termination of step 1).

   If there is no such set $\mathcal{E}'$ and no atom A, then proceed with Step 3.

2. Replace in $\mathcal{E}$ the elements in $\mathcal{E}'$ by A. More formally, $\mathcal{E} := (\mathcal{E} \setminus \mathcal{E}') \cup \{A\}$. Continue with Step 1.

3. Remove from $\mathcal{E}$ as many elements as necessary in order satisfy condition 3 in Definition 4.13.

4. Return $\mathcal{E}$.

The termination of this procedure follows from the fact that in step 2 the new element A is strictly greater than any maximal element in $\mathcal{E}'$ it replaces (wrt. the ordering $\gtrsim$). Moreover, it can be verified that a normalization of $\mathcal{E}$ always exists, and it is unique up to renaming of its members. This justifies to speak about "the" normalization of $\mathcal{E}$ in the sequel, and $\mathcal{E}^\nu$ is used to denote some uniquely determined normalization of $\mathcal{E}$.

Example 4.15 Let us consider an atom set $\mathcal{E} = \{P(a)\}$, where a is the only ground term. In this case a represents the whole domain, so $\mathcal{E}$ can be normalized to $\mathcal{E}^\nu = \{P(x)\}$.

Now we consider $\mathcal{E} = \{P(f(f(x))), P(f(a)), P(a), P(f(b))\}$, where a and b are the only constant function symbols and f is the only other function symbol. Since $P(b) \notin \mathcal{E}$, $\mathcal{E}$ itself cannot be normalized to $\{P(x)\}$. Instead we consider the subset $\mathcal{E}' = \{P(f(f(x))), P(f(a)), P(f(b))\}$, which leads to the normalization $\mathcal{E}^\nu = \{P(f(x)), P(a)\}$. □

As announced, using the normalization operation there is an easy test if a given AWE covers at least one ground atom:

Proposition 4.16
Let $A - \mathcal{E}$ be a an AWE, Σ a signature and $\mathcal{E}^\nu$ be the normalization of $\mathcal{E}$ wrt. Σ. Then, $A - \mathcal{E}$ covers no ground Σ-atom iff $A - \mathcal{E}^\nu$ is of the form $A - \{A'\}$, where $A' \approx A$.

For instance, if the signature under consideration Σ contains just the constant a, them from the AWE $P(y) - \{P(a)\}$ one obtains $P(y) - \{P(a)\}^\nu = P(y) - \{P(x)\}$, which indicates that $P(y) - \{P(a)\}$ does not cover any ground Σ-atom.

4.5 Computing with Atoms with Exceptions

The usual variant-relation "≈" among atoms is extended to AWEs in the following way.

Definition 4.17 (Variantship of AWEs)
We say that AWEs $A - \mathcal{E}$ and $A' - \mathcal{E}'$ are *variants*, and write $A - \mathcal{E} \approx A' - \mathcal{E}'$, iff

1. $A \approx A'$, and
2. there is a bijection from $\mathcal{E}$ onto $\mathcal{E}'$ such that each element from $\mathcal{E}$ is mapped to a variant of it in $\mathcal{E}'$.

□

In Section 4.2 above we introduced the $\in_\approx$-relation among atoms and sets of atoms. This relation is extended to AWEs $A - \mathcal{E}$ and sets of AWEs $\mathcal{N}$ by defining $A - \mathcal{E} \in_\approx \mathcal{N}$ iff $\mathcal{N}$ contains a variant of $A - \mathcal{E}$.

Vaguely spoken, reasoning with AWEs shall occur at the first-order level. To this end we extend the context of most general unifiers to the framework introduced so far. The following Definition 4.18 makes this idea precise.

Definition 4.18 (Branch Unifier)
Let $\mathcal{N}$ be a set of AWEs and $\mathcal{B} = \{B_1, \ldots, B_k, \textsf{not}\, B_{k+1}, \ldots, \textsf{not}\, B_n\}$ a set of atoms or default negative atoms. A pair of substitutions (σ, γ) is a *branch unifier of* $\mathcal{B}$ *against* $\mathcal{N}$ iff there are fresh variants $A_1 - \mathcal{E}_1, \ldots, A_n - \mathcal{E}_n \in_\approx \mathcal{N}$ and atoms $E_{k+1} \in \mathcal{E}_{k+1}, \ldots, E_n \in \mathcal{E}_n$ such that

1. σ is a most general simultaneous unifier of $\{A_1, B_1\}, \ldots, \{A_k, B_k\}$, $\{E_{k+1}, B_{k+1}\}, \ldots, \{E_n, B_n\}$, and
2. $A_i - \mathcal{E}_i$ is a MSC of $B_i\sigma\gamma$ in $\mathcal{N}$, for all $i = 1, \ldots, k$, and
3. $\mathcal{N}$ does not cover $B_i\sigma\gamma$, for all $i = k + 1, \ldots, n$.

We say that a single substitution σ alone is a branch unifier of $\mathcal{B}$ against $\mathcal{N}$ iff (σ, ϵ) is a branch unifier of $\mathcal{B}$ against $\mathcal{N}$.[11]
□

Example 4.19 (Branch Unifier) Let $\mathcal{N} = \{P(x, y) - \{P(x, x), P(a, b)\}\}$ be a set of AWEs and $\mathcal{B} = \{P(z, a), \textsf{not}\, P(a, z)\}$. Then, the substitution $\{x \mapsto b,\ y \mapsto a,\ z \mapsto b\}$ is a branch unifier of $\mathcal{B}$ against $\mathcal{N}$.[12] However, $\sigma = \{x \mapsto a,\ y \mapsto a,\ z \mapsto a\}$, which is a simultaneous most general unifier of $\{P(x, y), P(z, a)\}$

[11] By ϵ, the empty substitution, i.e. the identity function is meant.
[12] For simplicity of presentation no fresh variants are taken if not essential.

and $\{P(x,x), P(a,z)\}$, is not a branch unifier of $\mathcal{B}$ against $\mathcal{N}$, because $P(x,y) - \{P(x,x), P(a,b)\}$ is not a MSC of $P(z,a)\sigma = P(a,a)$ in $\mathcal{N}$ (it is not a cover at all).
□

Branch unifiers are a purely syntactical concept, and existence of branch unifiers against finite sets of AWEs obviously is decidable.

Branch unifiers will be employed in the calculus below to determine if a rule body of the form $\mathcal{B} = \{B_1, \ldots, B_k, \mathsf{not}\, B_{k+1}, \ldots, \mathsf{not}\, B_n\}$ is satisfiable in the interpretation $\mathcal{I}_{\mathcal{N}}$ associated to a current set of AWEs $\mathcal{N}$. To this end, candidate atoms $A_1, \ldots, A_k$ and $E_{k+1}, \ldots, E_n$ are chosen first, as mentioned in Definition 4.18 above. Then, a branch unifier σ is sought. However, just computing branch unifiers this way is not sufficient to get a correct calculus. One more concept is needed, which can be indicated as follows.

Recall that a set of atoms or default atoms $\mathcal{B}$ is satisfiable in $\mathcal{I}_{\mathcal{N}}$ iff some ground instance of $\mathcal{B}$ is *true* in $\mathcal{I}_{\mathcal{N}}$. Having this in mind, "branch unifiers" and "satisfiability" are related as follows: it is not too difficult to see that a ground atom B can be *true* in $\mathcal{I}_{\mathcal{N}}$ only if $\mathcal{N}$ contains an MSC of it; likewise, a ground default negative atom $\mathsf{not}\, B$ can be *true* in $\mathcal{I}_{\mathcal{N}}$ only if B is an instance of some atom in the exception part of some AWE in $\mathcal{N}$, *and if $\mathcal{N}$ does not cover B* (because otherwise B were *true* and so $\mathsf{not}\, B$ were *false*).[13]

Speaking imprecisely, after a branch unifier σ of $\mathcal{B}$ against $\mathcal{N}$ has been computed, one still needs to determine instances of $\mathcal{B}\sigma$, the satisfiability of which is violated in $\mathcal{I}_{\mathcal{N}}$. Because the members of $\mathcal{B}$ are connected conjunctively, it suffices to determine these instances *individually*. These considerations motivate the following definition.

Definition 4.20 (Branch-Exception Unifier)
Let $\mathcal{N}$ be a set of AWEs, B an atom and δ a substitution.

We say that δ is a *branch-exception unifier of B against $\mathcal{N}$* iff there is a $A - \mathcal{E} \in_{\approx} \mathcal{N}$ and an atom $E \in \mathcal{E}$ such that

1. δ is a most general unifier of $\{E, B\}$, and
2. $\mathcal{N}$ does not cover $B\delta$.

We say that δ is a *branch-exception unifier of* $\mathsf{not}\, B$ *against* $\mathcal{N}$ iff there is a $A - \mathcal{E} \in_{\approx} \mathcal{N}$such that

1. δ is a most general unifier of $\{A, B\}$, and
2. $A - \mathcal{E}$ is a MSC of $B\delta$ in $\mathcal{N}$.

[13] At least if $\mathcal{N}$ contains a pair $x - \{x\}$, as explained further below.

□

Observe that B and $\mathsf{not}\, B$ in this definition have the same rôle as the $\mathsf{not}\, B_i$'s and B_i's, respectively, in the definition of branch unifier above. Therefore, strictly speaking, the definition of branch exception unifier is redundant.

Example 4.21 (Branch-Exception Unifier) Consider the set $\mathcal{N} = \{P(x,y) - \{P(x,x), P(a,b)\}\}$ from Example 4.19 again. Then, both $\delta_1 = \{z \mapsto a\}$ and $\delta_2 = \{z \mapsto b\}$ are branch exception unifiers of $P(a,z)$ against N. The substitution $\delta_3 = \{x \mapsto a, y \mapsto z\}$ is a branch exception unifier of $\mathsf{not}\, P(a,z)$ against $\mathcal{N}$. There is no branch exception unifier of $\mathsf{not}\, P(z,z)$ against $\mathcal{N}$. □

4.6 Hyper Tableaux for Weak Perfect Models

In this section a Hyper Tableau calculus to compute weak perfect models will be introduced. Before doing so, one more preliminary step must be taken. It can be explained as follows.

It is clear that an atom A can be assigned *true* by a set of AWEs only if some member of A covers it. The "symmetrical" case, in the sense that an atom A can be assigned *false* only if it is an instance of some atom of some exception in $\mathcal{N}$, does not hold, however. For instance, $P(a)$ will be assigned *false* if $\mathcal{N}$ is empty. For technical reasons it is preferable to achieve the mentioned "symmetrical" case. To this end, we will add to $\mathcal{N}$ the pair $x - \{x\}$, for some variable x. Observe that $x - \{x\}$ is *not* an AWE, because $x \gtrsim\!\!\!\!/\; x$ does not hold. However, this will be the only "AWE" of this kind considered, and treating it as an AWE will not cause any problems. Having said this, the "AWE" $x - \{x\}$ now acts as a default assignment to *false* to atoms, and the exception part $\{x\}$ achieves the desired "symmetry".

From now on, the signatures under considerations are assumed to contain a 0-ary predicate symbol $\perp$ that is different from all other predicate symbols. Furthermore, the symbol $\perp$ is interpreted by the truth value "false". That is, no model of any clause set can contain $\perp$.

In the following, $\mathcal{N}$ always denotes a finite sequence of AWEs. We say that such a sequence $\mathcal{N}$ is *closed* iff it contains $\perp$ (or, more explicitly, if it contains $\perp - \{\}$), and we say that $\mathcal{N}$ is *open* iff it is not closed.

Where a set of AWEs is required, but a sequence of AWEs is given, the obvious conversion is implicitly assumed.

The KRHyper calculus consists of three inference rules. They allow to derive from a given sequence of AWEs and a given clause one or more than one sequence(s) of AWEs. There is one rule to close a sequence of AWEs $\mathcal{N}$ by an integrity constraint, one rule to extend $\mathcal{N}$ by means of a normal clause, and one rule to extend $\mathcal{N}$ in a branching way by means of a disjunctive rule.

Definition 4.22 (KRHYPER **Inference Rules)**
In the following, sequences of AWEs noted just $\mathcal{N}$ are assumed to be open.

$$\mathsf{Close}\ \frac{\mathcal{N} \qquad \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-}{(\mathcal{N}, \bot)} \quad \text{if } (*)$$

where (*): there is a branch unifier of the form (σ, ϵ) of $\mathcal{B}^+ \cup \mathcal{B}^-$ against $\mathcal{N}$.

We say that an application of the Close inference rule is *blocked (by regularity)* iff $\bot \in_\approx \mathcal{N}$.

$$\mathsf{Def\text{-}Ext}\ \frac{\mathcal{N} \qquad A_1 \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-}{(\mathcal{N}, A_1\sigma - \mathcal{E}^\nu)} \quad \text{if } (*)$$

where (*): (σ, ϵ) is a branch unifier of $\mathcal{B}^+ \cup \mathcal{B}^-$ against $\mathcal{N}$, and $\mathcal{E}$ is a smallest set of atoms satisfying the following conditions and such that $A_1\sigma \notin_\approx \mathcal{E}^\nu$:

(i) for every $B \in \mathcal{B}^+$, and for every branch-exception unifier δ of $B\sigma$ against $\mathcal{N}$: if $A_1\sigma \not\approx A_1\sigma\delta$, then $A_1\sigma\delta \in_\approx \mathcal{E}$.

(ii) for every $\mathsf{not}\, B \in \mathcal{B}^-$, and for every branch-exception unifier δ of $\mathsf{not}\, B\sigma$ against $\mathcal{N}$: if $A_1\sigma \not\approx A_1\sigma\delta$, then $A_1\sigma\delta \in_\approx \mathcal{E}$.

We say that an application of the Def-Ext inference rule is *blocked (by regularity)* iff $A_1\sigma - \mathcal{E}^\nu \in_\approx \mathcal{N}$.

$$\mathsf{Disj\text{-}Ext}\ \frac{\mathcal{N} \qquad A_1 \vee \cdots \vee A_m \leftarrow \mathcal{B}^+ \cup \mathcal{B}^-}{(\mathcal{N}, A_1\sigma\gamma) \quad \cdots \quad (\mathcal{N}, A_m\sigma\gamma)} \quad \text{if } (*)$$

where (*): $m > 1$, the pair (σ, γ) is a branch unifier of $\mathcal{B}^+ \cup \mathcal{B}^-$ against $\mathcal{N}$, and γ is a ground substitution for $(A_1 \vee \cdots \vee A_m)\sigma$.

We say that an application of the Disj-Ext inference rule is *blocked (by regularity)* iff $A_i\sigma\gamma \in_\approx \mathcal{N}$, for all i with $1 \leq i \leq m$. □

The requirement in the Def-Ext inference rule that $\mathcal{E}$ must be a smallest set guarantees that $\mathcal{E}$ does not contain variants of the same atom. Whenever $\mathcal{N}$ is finite, so will be $\mathcal{E}$ then.

The blocked-condition in the definition of the Disj-Ext might seem unusual, as it permits to branch on a clause, say, $A \vee B \leftarrow$ if A (but not B) is contained in the branch. However, the "usual" stronger condition insisting that $A_i\sigma\gamma \in_\approx \mathcal{N}$, for *some* i with $1 \leq i \leq m$, cannot be used in our case. Intuitively, the split of a program containing both, say, $A \leftarrow$ and $A \vee B \leftarrow$ might chose $B \leftarrow$, and so any weak perfect model of the program (which shall be computed by the calculus) must contain both A and B.

Example 4.23 Consider again $\mathcal{N} = \{P(x,y) - \{P(x,x), P(a,b)\}\}$ from previous examples. The Close inference rule is applicable to $\mathcal{N}$ and the clause $\leftarrow \{P(z,a), \text{not}\, P(a,z)\}$; the branch unifier used, $\{x \mapsto b,\ y \mapsto a,\ z \mapsto b\}$, was also given in Example 4.19 above.

The Def-Ext inference rule is applicable to $\mathcal{N}$ and the rule $Q(z) \leftarrow P(a,z)$ in the following way. First, a branch unifier of the rule body $P(a,z)$ against $\mathcal{N}$ is computed. The substitution $\sigma = \{x \mapsto a, y \mapsto z\}$ is one. Starting with $P(a,z)\sigma = P(a,z)$, branch exception unifiers of $P(a,z)$ against $\mathcal{N}$ are computed. There are two of those: $\delta_1 = \{x \mapsto a, z \mapsto a\}$ and $\delta_2 = \{z \mapsto b\}$. They lead to the new AWE $Q(z) - \{Q(z)\delta_1, Q(z)\delta_2\} = Q(z) - \{Q(a), Q(b)\}$. Now, the next step is to normalize its exception part (cf. Def. 4.14). The result depends from the signature.

Suppose first a signature that just contains the constants a and b. Then $\{Q(a), Q(b)\}^\nu = \{Q(x)\}$ (for some variable x). But then, since $Q(z) \in_\approx \{Q(x)\}$ holds, Def-Ext is not applicable to $\mathcal{N}$ and the rule $Q(z) \leftarrow P(a,z)$. That this is the result to be expected can be seen from the set of ground atoms that are covered by $\mathcal{N}$, which is just $\{P(b,a)\}$. Recall that such sets stands for interpretations, which is here the interpretation that assigns *true* to $P(b,a)$, but *false* to every other atom. Now, in all ground instances of the rule $Q(z) \leftarrow P(a,z)$, the rule body is *false* in this interpretation, and hence by this rule neither $Q(a)$ nor $Q(b)$ should be derivable. (As we have seen, we rule is indeed not applicable).

Now suppose a signature that contains the constants a, b and c. Then, $\{Q(a), Q(b)\}^\nu = \{Q(a), Q(b)\}$, and so $Q(z) \notin_\approx \{Q(a), Q(b)\}$ follows, and Def-Ext is applicable to $\mathcal{N}$ and the rule $Q(z) \leftarrow P(a,z)$. It will extend $\mathcal{N}$ by the AWE $Q(z) - \{Q(a), Q(b)\}$. Using a similar line of reasoning as above, it is not too difficult to see that this is the "expected" result.

There is one feature of the Def-Ext inference rule that has not been demonstrated so far. It is relevant when a rule contains variables in the head that do not occur in the body. Take for instance the rule $Q(z) \leftarrow P(y)$ and $\mathcal{N} = \{P(x) - \{P(a)\}\}$. To apply Def-Ext, the branch unifier $\sigma = \{x \mapsto y\}$ of $P(y)$ against $\mathcal{N}$ may be considered (or $\sigma = \{y \mapsto x\}$, which will behave the same in the following argumentation). It leads to the branch exception unifier $\delta = \{y \mapsto a\}$ of $P(y)$ against $\mathcal{N}$. Now, observe that $Q(z)\sigma = Q(z) = Q(z)\sigma\delta$, and so by Def-Ext the given set $\mathcal{N}$ can be extended by $Q(z) - \{\}$ (but not by $Q(z) - \{Q(z)\}$).

To see that this is the expected result, one has to suppose that the exception parts in every AWE in $\mathcal{N}$ are normalized. (Indeed, the calculus derives only sets of AWEs $\mathcal{N}$ of this kind). Hence, $\mathcal{N} = \{P(x) - \{P(a)\}\}$ covers at least one ground atom, say, $P(b)$. But then, the rule $Q(z) \leftarrow P(y)$ stands in particular for the instance $Q(z) \leftarrow P(b)$, which sanctions the derivation of $Q(z)$. Had we not given the normalization property as mentioned, it would be impossible to say if $\{P(x) - \{P(a)\}\}$ covers a ground atom, and so deriving $Q(z)$ might not be valid.

Finally, to give an example for the Disj-Ext inference rule, consider the disjunctive rule $P(y) \vee Q(y,z) \leftarrow R(y)$ and the set of AWEs $\mathcal{N} = \{R(x) - \{R(a)\},\ Q(b,b) - \{\}\}$. The pair (σ, γ), where $\sigma = \{y \mapsto x\}$ and $\gamma = \{x \mapsto a, z \mapsto b\}$, cannot be used for an Disj-Ext inference rule application, because $R(x) - \{R(a)\}$ is not a MSC of $R(y)\sigma\gamma = R(a)$ (it does not cover $R(a)$ at all). However, the pair (σ, γ), where $\sigma = \{y \mapsto x\}$ and $\gamma = \{x \mapsto b, z \mapsto b\}$ can be used for an Disj-Ext inference rule application. It will branch on the disjunction $P(b) \vee Q(b,b)$. Observe that blocking by regularity does not apply, although $Q(b,b) \in_{\approx} \mathcal{N}$ holds. In fact, the inference rule is expected to be applicable as shown, because, intuitively, the (perfect) model for the split program $P(b) \leftarrow R(b)$ shall not be missed. □

The main data structure of the KRHyper Tableau Calculus are trees, the nodes of which are labeled with AWEs. In the following definition and further below, where a sequence of AWE's is required, but a branch in such a tree is given, the branch stands for the sequence of its labels, in the order given by the branch.

Definition 4.24 (KRHyper **Tableau)**
Let P be a program, stratified via $P_1 \cup \cdots \cup P_n$. A KRHyper *Tableau for P* is a labeled tree inductively defined as follows:

1. A one-node tree is a derivation tree iff its root is labeled with a pair of the form $x - \{x\}$

   Any such tree is called an *initial* KRHyper-*tableau*.

2. A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to an open branch $\mathcal{N}$ in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the branches $(\mathcal{N}, N_1), \ldots, (\mathcal{N}, N_m)$ can be derived by applying an inference rule to the branch $\mathcal{N}$ and a clause $C \in P_i$, for some i with $1 \leq i \leq n$, and such that the following holds:

   (a) The inference rule application is not blocked. *(Regularity)*

   (b) For every j with $1 \leq j < i$ and every clause $D \in P_j$, if an inference rule is applicable to $\mathcal{N}$ and D, then this application is blocked. *(Conformance to Stratification)*

   In this case we say that $\mathbf{T}'$ *is derived from* $\mathbf{T}$.

A KRHyper-tableau is said to be *closed* iff each of its branches is closed. Otherwise it is *open*.
□

Notice that closed tableaux cannot be extended further. Notice further that branches are always finite, as tableaux are finite. Alternatively, KRHyper

tableaux could be defined as transfinite trees, where the limit cases are given by the transitions to the next stratum. We refrain from such a definition here, because it would be (slightly) more complicated, and for our purposes we are even interested in *finite* trees only.

In the sequel, the letter κ will denote an ordinal smaller than or equal to the first infinite ordinal. For every κ then, we will denote a (possibly infinite) sequence $a_0, a_1, a_2, \ldots$ of κ elements by $(a_i)_{i<\kappa}$.

Definition 4.25 (Derivation)
Let P be a program, stratified via $P_1 \cup \cdots \cup P_n$.

A *derivation (from a program P)* is a possibly infinite sequence of derivation trees $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$, such that $\mathbf{T}_0$ is an initial KRHyper-tableau, and for all i with $0 < i < \kappa$, $\mathbf{T}_i$ is derived from $\mathbf{T}_{i-1}$. □

We call a derivation from P a *refutation of P* iff it ends in a closed KRHyper-tableau. A finite derivation is *exhausted* iff it cannot be extended further. A *derivation of a* KRHyper*-tableaux* $\mathbf{T}$ is a finite derivation whose last element is $\mathbf{T}$.

We conclude the technical description of the calculus by stating our main result; a proof is beyond the scope of the this paper and will be published elsewhere.

Theorem 4.26 (KRHyper Tableau Compute Weak Perfect Models)
Let P be a stratified program and $\mathcal{D}$ be an exhausted (hence finite) derivation from P of a KRHyper*-tableau* $\mathbf{T}$. *Then, the following holds.*

Soundness: *for every open branch $\mathcal{N}$ in $\mathbf{T}$, $\mathcal{I}_\mathcal{N}$ is a weak perfect model of P.*

Completeness: *for every weak perfect model $\mathcal{I}$ of P, there is an open branch $\mathcal{N}$ in $\mathbf{T}$ such that $\mathcal{I}_\mathcal{N} = \mathcal{I}$.*

Notice that the theorem claims completeness not for all (stratified) programs – which would be impossible to have – but just for those cases where an exhausted derivation exists. A sufficient condition to enforce finite derivations is given by the restriction to function-free signatures (but constants are allowed). Even in this restricted case we claim that much shorter derivations can be achieved as would be possible via the usual approach of computing with ground programs.

4.7 Examples

To see how we use the calculus in the context of Living Books, we take an excerpt from the knowledge management component (Figure 5). Its purpose is to find out by means of the `known_unit_inferred` relation whether the user has knowledge

about some unit in question. The clauses from the excerpt are stratified, i.e. they are organized in a hierarchical way.

```
%% User knowledge:
known_unit(analysis/1/2/_ALL_).                                  (1)
unknown_unit(analysis/1/2/1).                                    (2)

%% Book meta data:
refers(analysis/1/2/3, analysis/1/0/4).                          (3)

%% 'known_unit' transitive closure:
known_unit(Book_B/Unit_B) :-                                     (4)
    known_unit(Book_A/Unit_A),
    refers(Book_A/Unit_A, Book_B/Unit_B).

%% 'unknown_unit' transitive closure
unknown_unit(Book_B/Unit_B) :-                                   (5)
    unknown_unit(Book_A/Unit_A),
    refers(Book_A/Unit_A, Book_B/Unit_B).

%% Derived:
known_unit_inferred(Book/Unit) :-                                (6)
    known_unit(Book/Unit),
    not unknown_unit(Book/Unit).
```

Figure 5: Excerpt from the knowledge base.

In clause (1), `_ALL_` is a universally quantified variable, which means that, intentionally, all sub-units of `analysis/1/2` are declared to be known. Clause (2) expresses the fact that the user does not know the unit `1/2/1` from the `analysis`-book. Clause (6) resolves the apparent inconsistency behind the just given explanation of clauses (1) and (2). It says that `Book/Unit` is contained in the `known_unit_inferred`-relation (the relation we are interested in), if it is "known" (by means of the `known_unit(Book/Unit)` declaration) *and* this circumstance is not overridden by an explicit "unknown"-declaration of the same unit (by means of `not unknown_unit(Book/Unit)`). This is the actual, pragmatic semantic of clauses (1) and (2): "unknown"-declarations are stronger than "known"-declarations. We think that this is appropriate modeling, as "unknown" units are never withhold from the user. Clauses (4) and (5) are expressing knowledge about the meta data relations "known_unit" and "unknown_unit".

Now we will demonstrate how our calculus works on the given clauses. For

sake of notational simplicity we will write `ku` for `known_unit`, `uu` for `unknown_unit`, `ki` for `known_unit_inferred`, `r` for `refers` and we will omit the `Book` variable.

We start with an initial KRHyper-tableau $\mathcal{N}$. First, we apply Def-Ext to (1), (3) and (4). By combining these clauses, the unit `analysis/1/0/4` is derived to be "known":

$$\frac{\texttt{ku(1/2/_ALL_)},\texttt{r(1/2/3, 1/0/4)} \qquad \texttt{ku(Unit_B)} \leftarrow \texttt{ku(Unit_A)},\texttt{r(Unit_A, Unit_B)}}{(\mathcal{N},\texttt{ku(1/0/4)})}$$

The resulting AWE has an empty exception set and is obtained with the branch unifier $\sigma = \{\texttt{Unit_A} \mapsto \texttt{1/2/_ALL_}, \texttt{Unit_B} \mapsto \texttt{1/0/4}\}$.

Next, we apply Def-Ext to (1),(2) and (6). The derivation says that in the `known_unit_inferred`-relation, all sub-units of `analysis/1/2`, with the exception of `analysis/1/2/1`, are known. Observe that the mentioned apparent contradiction between what clauses (1) and (2) say is eliminated as explained.

$$\frac{\texttt{ku(1/2/_ALL_)},\texttt{uu(1/2/1)} \qquad \texttt{ki(Unit)} \leftarrow \texttt{ku(Unit)},\mathsf{not}\,\texttt{uu(Unit)}}{(\mathcal{N},\texttt{ki(1/2/_ALL_)} - \{\texttt{ki(1/2/1)}\})}$$

The resulting AWE is calculated in two steps: first, we find a branch unifier, $\sigma = \{\texttt{Unit} \mapsto \texttt{1/2/_ALL_}\}$. The instantiated head atom $\texttt{ki(Unit)}\sigma$ is `ki(1/2/_ALL_)`. In the second step we calculate the exception set. With the branch-exception unifier $\delta = \{\texttt{_ALL_} \mapsto 1\}$ of $\mathsf{not}\,\texttt{uu(1/2/_ALL_)}$ against $\mathcal{N}$ we get as only exception `ki(1/2/1)`. The resulting AWE, which is already normalized, is then added to our tableau.

Figure 6 shows the hyper tableau for the preceding calculus steps.

```
known_unit(analysis/1/2/_ALL_)
unknown_unit(analysis/1/2/1)
refers(analysis/1/2/3, analysis/1/0/4)
                 |
known_unit(analysis/1/0/4)
                 |
known_unit_inferred(analysis/1/2/_ALL_)
- { known_unit_inferred(analysis/1/2/1) }
```

Figure 6: Hyper tableau derivation from the clauses in Figure 5.

5 Conclusions

In this article, we have described the Living Book and the use of theorem proving technology as a core component in the knowledge management system (KMS) of the Living Book. The task of the KMS is to assemble new documents from a database of elementary units representing the given learning material with respect to various learning scenarios. The computation of such assemblies is carried out by a model-generating theorem prover for first-order logic with a default negation principle. Its input consists of meta data that describe the dependencies between different elementary units, and logic-programming style rules that describe the scenario composition of these units. Additionally, a user model is taken into account that contains information about topics and slices that are known or unknown to a learner. A model computed by the system for such input then directly specifies the document to be assembled.

The deduction technique used in the KMS is based on the KRHyper deduction system. In this work, we have shown that the deduction techniques as realized in KRHyper is a suitable means for combining various kinds of knowledge given by the various learning resources, scenarios and user profiles in order to derive adaptive personalized documents within the e-learning context.

The paradigm used to describe the computations is a rather general one: first-order logic programs with a default negation operator. Furthermore, parts of the programs specify computations that are not connected to the E-learning domain *as such*, but are equally usable in different contexts as well. For instance, the system uses a binary relation “refers” between units and computes its transitive closure. Obviously, such “reference” relations occur frequently between all types of information in various domains . In conclusion, beyond having demonstrated that our Living Book system is an example for a successful use of deduction techniques in a complex real-world application, we think that the underlying techniques can readily be employed for a rather wide class of applications

The calculus of the deduction component in the Living Book has been the focus of this work. Its main aspects and our contribution with respect to this calculus is summarized in the next section and discussed in the context of other approaches in section 5.2.

5.1 Calculus

One of the big challenges in both classical logic and non-monotonic 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 designing respective decision procedures, roughly, that detect and prune loops in derivations (see e.g.

[16, 30, 3, 36]) or restrict the input language such that loops cannot occur at all [19, e.g.]. Our approach is of the latter kind.

In the field of logic programming, a common viewpoint is to identify a program with the set of all its ground instances and then to apply propositional methods (see below for first-order methods) . Of course, this approach is feasible only in restricted cases, when reasoning can be restricted to a finite subset of the possibly infinite set of ground instances. Even the best systems following this approach reach their limits when confronted with sufficiently large data sets.

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 seems not a viable option. Therefore, our KRHyper tableau 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. As a starting point we took a calculus originally developed for first-order classical reasoning – hyper tableaux [6] – and modified it according to the weak perfect model semantics.

Only some approaches have been reported in the literature that also work on the first-order level. For the well-founded semantics and the stable model semantics of *normal* programs see [22, 21], and for the well-founded semantics for disjunctive programs see [14]. The closest relative to our approach is the bottom-up method to compute perfect models of (stratified) disjunctive programs in [17]. The "model trees" defined there are structurally identical to our KRHyper tableaux, however, unlike KRHyper Tableaux, they work on the *ground* level. For the common domain of propositional programs, the KRHyper tableau approach could be refined to obtain the model tree approach by adding a "model minimality test" that would close a branch after a stratum has been exhausted and the resulting interpretation is not a minimal model for the underlying split program up to the considered stratum. For the general case of non-propositional programs it will be future work to see if such a model minimality test can be added to KRHyper tableaux, so that perfect models instead of weak perfect models will be computed.

5.2 Other Approaches

In this work, specifications in the logic programming style are advocated as an appropriate formalism to model our task at hand. In brief, our approach is completely declarative, and hence we believe that there are advantageous over more procedural oriented techniques. 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, e.g., `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 our programs (like eg. in Section 3.3). Furthermore, 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 few 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* [33]. XSB-Prolog supports query answering wrt. the well-founded semantics for normal logic programs [38]. 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 solutions.

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 [35, e.g.] with greatly enhanced expressive power exist, and efficient respective systems to reason about DL specifications have been developed [25].

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.

At the current state of our work, however, it is not yet clear how we can implement disjunctive and non-monotonic reasoning 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.

5.3 Status of This Work and Perspectives

The calculus and its implementation are developed far enough, so that practical use is 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 [24] is integrated. Without indexing, even moderately sized problems are not solvable. With indexing, the response time for a typical query with a database stemming from about 100 units takes less than a second. A similar query, applied to a richer book with about 4000 units takes ten seconds, which seems almost acceptable. Currently, the Living Book system manages the units of one book 'Logic for Computer Scientists'; we already have gained experience with the combination of learning material of multiple books but we still think that some optimization is necessary in order to achieve the required performance for this amount of data. An optimized implementation of the KRHyper system is currently developed [41].

Related to the functionality of the Living Book system and the e-learning context there are two more aspects we want to address. Firstly, the scenario management presented so far considers the various scenarios as independent from each other. It would be useful from a pedagogical point of view, though, to relate the various scenarios and to specify dependencies between them. This could be used by a teacher in order to define various *learning paths* for students.

Secondly, a more advanced aspect that we want to address aims at collaborative learning. In collaborative learning, a group of students is working on the solution of a problem. The group is supposed to be guided by experts in group behavior and pedagogical aspects of collaborative learning. The idea is to formalize the domain knowledge of these experts within our logic-based framework and to compute models representing the relevant learning material for such a group.

Acknowledgements. We are grateful to Christoph Wernhard for valuable discussions about the calculus.

References

[1] Bachmair, L., and Ganzinger, H. Perfect Model Semantics for Logic Programs with Equality. In *Proc. ICLP* (1991).

[2] BAUMGARTNER, P. Hyper Tableaux — The Next Generation. In *Automated Reasoning with Analytic Tableaux and Related Methods* (1998), H. de Swaart, Ed., vol. 1397 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 60–76.

[3] BAUMGARTNER, P. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In *CADE-17 – The 17th International Conference on Automated Deduction* (2000), D. McAllester, Ed., vol. 1831 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 200–219.

[4] BAUMGARTNER, P. Automated deduction techniques for the management of personalized documents. In *Proc. of the IJCAR-Workshop* Future Directions in Automated Reasoning (Siena, Italy, 2001), M. Kerber, Ed.

[5] BAUMGARTNER, P., AND FURBACH, U. Automated Deduction Techniques for the Management of Personalized Documents. *Annals of Mathematics and Artificial Intelligence – Special Issue on Mathematical Knowledge Management* (Kluwer Academic Publishers, 2002). Accepted for Publication.

[6] BAUMGARTNER, P., FURBACH, U., AND NIEMELÄ, I. Hyper Tableaux. In *Proc. JELIA 96* (1996), no. 1126 in Lecture Notes in Artificial Intelligence, European Workshop on Logic in AI, Springer.

[7] BAUMGARTNER, P., GROSS-HARDT, M., AND SIMON, A. B. Living Book – An Interactive and Personalized Book. In *SSGRR 2002s - International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet* (2002), V. Milutinovic, Ed., Published electronically (http://www.ssgrr.it/en/ssgrr2002s/papers.htm).

[8] BREWKA, G., DIX, J., AND KONOLIGE, K. *Nonmonotonic Reasoning*, vol. 73 of *Lecture Notes*. CSLI Publications, 1997.

[9] BRY, F., AND KRAUS, M. Perspectives for electronic books in the world wide web age. *The Electronic Library Journal 20*, 4 (2002), 275–287.

[10] CHANG, C., AND LEE, R. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[11] DAHN, I. Slicing book technology - providing online support for textbooks. In *Proc. of the 20th World Conference on Open and Distance Learning* (Düsseldorf/Germany, 2001), H. Hoyer, Ed.

[12] DAHN, I. Using networks for advanced personalization of documents. In *Proc. SSGRR 2001* (L'Aquila/Italy, 2001), V. Milutinovic, Ed.

[13] Dix, J., Furbach, U., and Niemelä, I. Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations. In *Handbook of Automated Reasoning*, A. Voronkov and A. Robinson, Eds. Elsevier-Science-Press, 2001, pp. 1121–1234.

[14] Dix, J., and Stolzenburg, F. A framework to incorporate non-monotonic reasoning into constraint logic programming. *Journal of Logic Programming 37*, 1-3 (1998), 47–76. Special Issue on *Constraint Logic Programming*. Guest editors: Kim Marriott and Peter J. Stuckey.

[15] Eder, E. Properties of Substitutions and Unifications. *Journal of Symbolic Computation 1*, 1 (March 1985).

[16] Fermller, C., and Leitsch, A. Hyperresolution and automated model building. *Journal of Logic and Computation 6*, 2 (1996), 173–230.

[17] Fernández, J. A., and Minker, J. Bottom-up computation of perfect models for disjunctive theories. *Journal of Logic Programming 25*, 1 (1995).

[18] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming, Seattle* (1988), R. Kowalski and K. Bowen, Eds., pp. 1070–1080.

[19] Georgieva, L., Hustadt, U., and Schmidt, R. A. Hyperresolution for guarded formulae. *Journal of Symbolic Computation* (2002). To appear.

[20] Goller, C., Letz, R., Mayr, K., and Schumann, J. Setheo v3.2: Recent developments — system abstract —. In *Automated Deduction — CADE 12* (Nancy, France, June 1994), A. Bundy, Ed., LNAI 814, Springer-Verlag, pp. 778–782.

[21] Gottlob, G., Marcus, S., Nerode, A., Salzer, G., and Subrahmanian, V. S. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science 166*, 1-2 (1996), 221–262.

[22] Gottlob, G., Marcus, S., Nerode, A., and Subrahmanian, V. Nonground stable and wellfounded semantics, 1994.

[23] Gottlob, G., and Pichler, R. Working with arms: Complexity results on atomic representations of herbrand models. In *Proceedings of the 14th Symposium on Logic in Computer Science* (1998), IEEE.

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

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

[26] In2Math. Interactive elements in mathematics and computer science education for undergraduates. `www.in2math.de`.

[27] Letz, R. Clausal Tableaux. In *Automated Deduction. A Basis for Applications* (1998), W. Bibel and P. H. Schmitt, Eds., Kluwer Academic Publishers.

[28] Manthey, R., and Bry, F. SATCHMO: a theorem prover implemented in Prolog. In *Proceedings of the 9th Conference on Automated Deduction, Argonne, Illinois, May 1988* (1988), E. Lusk and R. Overbeek, Eds., vol. 310 of *Lecture Notes in Computer Science*, Springer, pp. 415–434.

[29] Melis, E., Andres, E., Büdenbender, J., Frischauf, A., Goguadze, G., Libbrecht, P., Pollet, M., and Ullrich, C. Activemath: A generic and adaptive web-based learning environment. *Journal of Artificial Intelligence and Education 12*, 4 (2001), 385–407.

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

[31] Principe, J. C., Euliano, N. R., and Lefebvre, W. C. Innovating adaptive and neural systems instruction with interactive electronic books. *Proc. IEEE 88*, 1 (2000), 81–95.

[32] Robinson, J. A. Automated deduction with hyper-resolution. *Internat. J. Comput. Math. 1* (1965), 227–234.

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

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

[35] Schmidt-Schau, M., and Smolka, G. Attributive concept descriptions with complements. *Artificial Intelligence 48*, 1 (1991), 1–26.

[36] Stolzenburg, F. Loop-detection in hyper-tableaux by powerful model generation. *Journal of Universal Computer Science 5*, 3 (1999), 135–155. Special Issue on *Integration of Deduction Systems*. Guest editors: Reiner

Hähnle, Wolfram Menzel, Peter H. Schmitt and Wolfgang Reif. Springer, Berlin, Heidelberg, New York.

[37] Trial-Solution. Project supported by european commission. www.trial-solution.de.

[38] Van Gelder, A., Ross, K. A., and Schlipf, J. S. The well-founded semantics for general logic programs. *Journal of the ACM 38* (1991), 620–650.

[39] Vassileva, J. Dynamic courseware generation at the www. In *Proc. of the 8th World Conference on AI and Education (AIED'97)* (Kobe, Japan, 1997).

[40] Weidenbach, C., Afshordel, B., Brahm, U., Cohrs, C., Engel, T., Keen, E., Theobalt, C., and Topić, D. System description: Spass version 1.0.0. In *CADE-16 – The 16th International Conference on Automated Deduction* (Trento, Italy, 1999), H. Ganzinger, Ed., vol. 1632 of *Lecture Notes in Artificial Intelligence*, Springer, pp. 378–382.

[41] Wernhard, C. System description: Krhyper3. Submitted, 2003.

Available Research Reports (since 1998):

2003

2/2003 *Peter Baumgartner, Margret Groß-Hardt, Alex Sinner.* Living Book – Deduction, Slicing and Interaction.

1/2003 *Peter Baumgartner, Cesare Tinelli.* The Model Evolution Calculus.

2002

12/2002 *Kurt Lautenbach.* Logical Reasoning and Petri Nets.

11/2002 *Margret Groß-Hardt.* Processing of Concept Based Queries for XML Data.

10/2002 *Hanno Binder, Jérôme Diebold, Tobias Feldmann, Andreas Kern, David Polock, Dennis Reif, Stephan Schmidt, Frank Schmitt, Dieter Zöbel.* Fahrassistenzsystem zur Unterstützung beim Rückwärtsfahren mit einachsigen Gespannen.

9/2002 *Jürgen Ebert, Bernt Kullbach, Franz Lehner.* 4. Workshop Software Reengineering (Bad Honnef, 29./30. April 2002).

8/2002 *Richard C. Holt, Andreas Winter, Jingwei Wu.* Towards a Common Query Language for Reverse Engineering.

7/2002 *Jürgen Ebert, Bernt Kullbach, Volker Riediger, Andreas Winter.* GUPRO – Generic Understanding of Programs, An Overview.

6/2002 *Margret Groß-Hardt.* Concept based querying of semistructured data.

5/2002 *Anna Simon, Marianne Valerius.* User Requirements – Lessons Learned from a Computer Science Course.

4/2002 *Frieder Stolzenburg, Oliver Obst, Jan Murray.* Qualitative Velocity and Ball Interception.

3/2002 *Peter Baumgartner.* A First-Order Logic Davis-Putnam-Logemann-Loveland Procedure.

2/2002 *Peter Baumgartner, Ulrich Furbach.* Automated Deduction Techniques for the Management of Personalized Documents.

1/2002 *Jürgen Ebert, Bernt Kullbach, Franz Lehner.* 3. Workshop Software Reengineering (Bad Honnef, 10./11. Mai 2001).

2001

13/2001 *Annette Pook.* Schlussbericht "FUN - Funkunterrichtsnetzwerk".

12/2001 *Toshiaki Arai, Frieder Stolzenburg.* Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing.

11/2001 *Kurt Lautenbach.* Reproducibility of the Empty Marking.

10/2001 *Jan Murray.* Specifying Agents with UML in Robotic Soccer.

9/2001 *Andreas Winter.* Exchanging Graphs with GXL.

8/2001 *Marianne Valerius, Anna Simon.* Slicing Book Technology — eine neue Technik für eine neue Lehre?.

7/2001 *Bernt Kullbach, Volker Riediger.* Folding: An Approach to Enable Program Understanding of Preprocessed Languages.

6/2001 *Frieder Stolzenburg.* From the Specification of Multiagent Systems by Statecharts to their Formal Analysis by Model Checking.

5/2001 *Oliver Obst.* Specifying Rational Agents with Statecharts and Utility Functions.

4/2001 *Torsten Gipp, Jürgen Ebert.* Conceptual Modelling and Web Site Generation using Graph Technology.

3/2001 *Carlos I. Chesñevar, Jürgen Dix, Frieder Stolzenburg, Guillermo R. Simari.* Relating Defeasible and Normal Logic Programming through Transformation Properties.

2/2001 *Carola Lange, Harry M. Sneed, Andreas Winter.* Applying GUPRO to GEOS – A Case Study.

1/2001 *Pascal von Hutten, Stephan Philippi.* Modelling a concurrent ray-tracing algorithm using object-oriented Petri-Nets.

2000

8/2000 *Jürgen Ebert, Bernt Kullbach, Franz Lehner (Hrsg.).* 2. Workshop Software Reengineering (Bad Honnef, 11./12. Mai 2000).

7/2000 *Stephan Philippi.* AWPN 2000 - 7. Workshop Algorithmen und Werkzeuge für Petrinetze, Koblenz, 02.-03. Oktober 2000 .

6/2000 *Jan Murray, Oliver Obst, Frieder Stolzenburg.* Towards a Logical Approach for Soccer Agents Engineering.

5/2000 *Peter Baumgartner, Hantao Zhang (Eds.).* FTP 2000 – Third International Workshop on First-Order Theorem Proving, St Andrews, Scotland, July 2000.

4/2000 *Frieder Stolzenburg, Alejandro J. García, Carlos I. Chesñevar, Guillermo R. Simari.* Introducing Generalized Specificity in Logic Programming.

3/2000 *Ingar Uhe, Manfred Rosendahl.* Specification of Symbols and Implementation of Their Constraints in JKogge.

2/2000 *Peter Baumgartner, Fabio Massacci.* The Taming of the (X)OR.

1/2000 *Richard C. Holt, Andreas Winter, Andy Schürr.* GXL: Towards a Standard Exchange Format.

1999

10/99 *Jürgen Ebert, Luuk Groenewegen, Roger Süttenbach.* A Formalization of SOCCA.

9/99 *Hassan Diab, Ulrich Furbach, Hassan Tabbara.* On the Use of Fuzzy Techniques in Cache Memory Managament.

8/99 *Jens Woch, Friedbert Widmann.* Implementation of a Schema-TAG-Parser.

7/99 *Jürgen Ebert, and Bernt Kullbach, Franz Lehner (Hrsg.).* Workshop Software-Reengineering (Bad Honnef, 27./28. Mai 1999).

6/99 *Peter Baumgartner, Michael Kühn.* Abductive Coreference by Model Construction.

5/99 *Jürgen Ebert, Bernt Kullbach, Andreas Winter.* GraX – An Interchange Format for Reengineering Tools.

4/99 *Frieder Stolzenburg, Oliver Obst, Jan Murray, Björn Bremer.* Spatial Agents Implemented in a Logical Expressible Language.

3/99 *Kurt Lautenbach, Carlo Simon.* Erweiterte Zeitstempelnetze zur Modellierung hybrider Systeme.

2/99 *Frieder Stolzenburg.* Loop-Detection in Hyper-Tableaux by Powerful Model Generation.

1/99 *Peter Baumgartner, J.D. Horton, Bruce Spencer.* Merge Path Improvements for Minimal Model Hyper Tableaux.

1998

24/98 *Jürgen Ebert, Roger Süttenbach, Ingar Uhe.* Meta-CASE Worldwide.

23/98 *Peter Baumgartner, Norbert Eisinger, Ulrich Furbach.* A Confluent Connection Calculus.

22/98 *Bernt Kullbach, Andreas Winter.* Querying as an Enabling Technology in Software Reengineering.

21/98 *Jürgen Dix, V.S. Subrahmanian, George Pick.* Meta-Agent Programs.

20/98 *Jürgen Dix, Ulrich Furbach, Ilkka Niemelä .* Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations.

19/98 *Jürgen Dix, Steffen Hölldobler.* Inference Mechanisms in Knowledge-Based Systems: Theory and Applications (Proceedings of WS at KI '98).

18/98 *Jose Arrazola, Jürgen Dix, Mauricio Osorio, Claudia Zepeda.* Well-behaved semantics for Logic Programming.

17/98 *Stefan Brass, Jürgen Dix, Teodor C. Przymusinski.* Super Logic Programs.

16/98 *Jürgen Dix.* The Logic Programming Paradigm.

15/98 *Stefan Brass, Jürgen Dix, Burkhard Freitag, Ulrich Zukowski.* Transformation-Based Bottom-Up Computation of the Well-Founded Model.

14/98 *Manfred Kamp.* GReQL – Eine Anfragesprache für das GUPRO–Repository – Sprachbeschreibung (Version 1.2).

12/98 *Peter Dahm, Jürgen Ebert, Angelika Franzke, Manfred Kamp, Andreas Winter.* TGraphen und EER-Schemata – formale Grundlagen.

11/98 *Peter Dahm, Friedbert Widmann.* Das Graphenlabor.

10/98 *Jörg Jooss, Thomas Marx.* Workflow Modeling according to WfMC.

9/98 *Dieter Zöbel.* Schedulability criteria for age constraint processes in hard real-time systems.

8/98 *Wenjin Lu, Ulrich Furbach.* Disjunctive logic program = Horn Program + Control program.

7/98 *Andreas Schmid.* Solution for the counting to infinity problem of distance vector routing.

6/98 *Ulrich Furbach, Michael Kühn, Frieder Stolzenburg.* Model-Guided Proof Debugging.

5/98 *Peter Baumgartner, Dorothea Schäfer.* Model Elimination with Simplification and its Application to Software Verification.

4/98 *Bernt Kullbach, Andreas Winter, Peter Dahm, Jürgen Ebert.* Program Comprehension in Multi-Language Systems.

3/98 *Jürgen Dix, Jorge Lobo*. Logic Programming and Nonmonotonic Reasoning.

2/98 *Hans-Michael Hanisch, Kurt Lautenbach, Carlo Simon, Jan Thieme*. Zeitstempelnetze in technischen Anwendungen.

1/98 *Manfred Kamp*. Managing a Multi-File, Multi-Language Software Repository for Program Comprehension Tools — A Generic Approach.