D4.5: Reasoning Engine – Version 2

Version 1.0 Final

S. Espinosa, A. Kaya, S. Melzer, R. Möller, T. Näth, M. Wessel

Hamburg University of Technology (TUHH)

Project ref.no.:	*FP6-027538*
Workpackage:	*WP4*
WP / Task responsible:	*TUHH*
Other contributors:	*none*
Quality Assurance:	*Kalliope Dalakleidi, Alfio Ferrara, Davide Lorusso*
Document type:	*Report*
Security (distribution level):	*Public*
Status & version:	*Final 1.0*
Contractual date of delivery:	*M17*
Actual date of delivery:	*7th-Aug-07*
Deliverable number:	*D4.5*
Deliverable name:	*Reasoning Engine – Version 2*
Date:	*7th-Aug-07*
Number of pages:	*55*
EC Project Officer:	*Johan Hagman*
Keywords:	*Description Logics, Reasoning Services, Reasoning under uncertainty*
Abstract (for dissemination):	*This report describes the most important extensions to the reasoning engine RacerPro which is used in the Boemie project. These extensions are targeted to support media interpretation, for dealing with learning, to sustain scalability issues, and to provide an initial basis for dealing with probabilistic knowledge in media interpretation.*

Executive Summary

This report describes the most important extensions to the reasoning engine RacerPro which is used in the Boemie project. These extensions are targeted to support media interpretation, for dealing with learning, to sustain scalability issues, and to provide an initial basis for dealing with probabilistic knowledge in media interpretation. In particular, we describe:

- Operators for supporting abduction for interpretation of multimedia documents: Given a knowledge base (Tbox, Abox, rules) and a formula F which should be entailed by the knowledge base, the abduction component derives a set of hypotheses which cannot be proven to hold, but – according to the rules – have to be assumed to be true in order to ensure that F is entailed.
- Event recognition facilities: Based on so-called event-rules, the reasoner can detect whether a temporal high-level event can be instantiated given a set of temporal propositions about basic events.
- Nonstandard inferences to support learning in Boemie: Racer now supports the least-common subsumer operator for the description logic $\mathcal{ALE}$ and can describe Abox individuals using an approximation of the most-specific concept (approximation up to nesting level k).
- Extensions to the RacerPro Abox query language: The query language of Racer is extended with facilities for server-side programming to support post-processing of query results in the description logic inference system.
- Support for reasoning with triples in secondary memory: In order to achieve scalability in various respects, Racer now allows for queries referring to triples stored in a so-called persistent triple-store.
- An extension to RacerPro for reasoning about probabilistic knowledge: In order to deal with uncertainty, a new kind of Tbox axiom is supported: so-called *conditional constraints*. Using this construct it is, for instance, possible to represent attribute uncertainty and structural uncertainty.

We provide examples that demostrate how the facilities can be used in the Boemie project, and evaluate many of the implemented techniques in the Boemie context.

Parts of this report have been published in workshops and conferences. Namely, Chapter 1 is published in [10] and [27]. The evaluation of the abduction framework is presented in [35]. The event recognition facility is described in [27]. The triple store reasoning facility was presented at the industrial conference SemTech-07. The probabilistic extension to Racer is described in detail in [28].

Contents

Chapter 1

Introduction

The reasoning engine used in the Boemie project (RacerPro) has been significantly extended in order to meet the requirements of the project. This report describes the most important extensions, which are (i) targeted to support media interpretation, (ii) deal with learning requirements, (iii) sustain scalability issues, and (iv) provide an initial basis for dealing with probabilistic knowledge in media interpretation. The reports is subdivided into different chapters which describe the main facilities added to RacerPro:

- Operators for supporting abduction for interpretation of multimedia documents: Abduction is based on rules, i.e., the set of rules specifies the set of abducibles. Given a knowledge base (Tbox, Abox, rules) and a formula F which should be entailed by the knowledge base, the abduction component derives a set of hypotheses which cannot be proven to hold, but – according to the rules – have to be assumed to be true in order to ensure that F is entailed. In the abduction process, rules are applied in a backward-chaining way.
- Event recognition facilities: Based on so-called event-rules, the reasoner can detect whether a temporal high-level event can be instantiated given a set of temporal proposition about basic events. Basic events are detected by, for instance, video analysis processes. High-level events can be used to derive high-level interpretations of multimedia documents.
- Nonstandard inferences to support learning in Boemie: Racer now supports the least-common subsumer operator for the description logic $\mathcal{ALE}$ and can describe Abox individuals using an approximation of the most-specific concept (approximation up to nesting level k).
- Extensions to the RacerPro Abox query language nRQL (new Racer Query Language: pronounce as "niracle" and hear it as "miracle"): The query language of Racer is extended with facilities for server-side programming to support post-processing of query results in the description logic inference system. In addition, head-projection and aggregation operators are defined that support frequently used post-processing requirements. Furthermore, combined Tbox and Abox queries can now be specified.
- Support for reasoning with triples in secondary memory: In order to achieve scalability in various respects, Racer now allows for queries referring to triples stored in a so-called persistent triple-store. Queries can be answered with and without reasoning. The latter is possible also on secondary memory.
- An extension to RacerPro for reasoning about probabilistic knowledge: In order to deal with uncertainty, a new kind of Tbox axiom is supported: so-called *conditional constraints*.

Using this construct it is, for instance, possible to represent attribute uncertainty and structural uncertainty. In addition, new kinds of inference services related to probabilistic knowledge are defined: probabilistic subsumption, probabilistic instance checking.

We provide examples that demostrate how the facilities can be used in the Boemie project, and evaluate many of the implemented techniques in the Boemie context.

Chapter 2

Abduction

Automated extraction of information from different types of multimedia documents such as image, text, video, and audio becomes more and more relevant for intelligent retrieval systems. An intelligent retrieval system is a system with a knowledge base and capabilities that can be used to establish connections between a request and a set of data based on the high-level semantics of the data (which can also be documents). Typically, nowadays, automated semantics extraction from multimedia occurs by using low-level features and is often limited to the recognition of isolated items if even. Examples are single objects in an image, or single words (or maybe phrases) in a text. However, multimedia documents such as images usually present more than objects detectable in a bottom-up fashion. For instance, an image may illustrate an abstract concept such as an event. An event in a still image can hardly be perceived without additional high-level knowledge.

We see multimedia interpretation as abduction (reasoning from effects to causes) in that we reason from observations (effects) to explanations (causes). The aim of this work is to present a novel approach for multimedia interpretation through Abox abduction, which we consider as a new type of non-standard retrieval inference service in DLs. In particular, we focus on the use of DL-safe-like rules for finding explanations and introduce a preference measure for selecting 'preferred' explanations.

2.1 Related Work on Media Interpretation and Abduction

The idea of formalizing interpretation as abduction is investigated in [16] in the context of text interpretation. In [38], Shanahan presents a formal theory of robot perception as a form of abduction. In this work, low-level sensor data is transformed into a symbolic representation of the world in first-order logic and abduction is used to derive explanations. In the context of scene interpretation, recently, in [29] the use of DLs for scene interpretation processes is described.

In this report we present a novel approach based on the combination of the works in [16, 38] and [29], and indicate how formal representation and reasoning techniques can be used for interpretation of information extracted from multimedia documents. The approach used description logics and rules, with abduction implemented with backward-chaining applied to the rules. In contrast to approaches such as [18], which use abduction in the context of rules in logic programming, we use description-logic reasoning for proving subgoals of (non-recursive) rules. Other approaches for abduction in description logics (e.g., [4]) have dealt with concept abduction only. In [9] among other abductive reasoning tasks in DLs also Abox abduction is discussed. A solution to

the Abox abduction problem is formally presented, but it is not shown how to derive solutions.

Abduction is investigated for supporting information retrieval based on high-level descriptions on media content. The approach builds on [26] and, in contrast to later related work such as [7], the approach is integrated into a mainstream description logic system and is based on high-level descriptions of media content.

2.2 Retrieval Inference Services

Before introducing abduction as a new inference service, we start with an overview of retrieval inference services that are supported by state-of-the-art DL reasoners.

The *retrieval* inference problem w.r.t. a Tbox $\mathcal{T}$ is to find all individuals mentioned in an Abox $\mathcal{A}$ that are instances of a certain concept C: $\{x \text{ mentioned in } \mathcal{A} \mid (\mathcal{T}, \mathcal{A}) \models x : C\}$. In addition to the basic retrieval inference service, expressive query languages are required in practical applications. Well-established is the class of conjunctive queries. A *conjunctive query* consists of a *head* and a *body*. The head lists variables for which the user would like to compute bindings. The body consists of query atoms (see below) in which all variables from the head must be mentioned. If the body contains additional variables, they are seen as existentially quantified. A query answer is a set of tuples representing bindings for variables mentioned in the head. A query is a structure of the form $\{(X_1, \ldots, X_n) \mid atom_1, \ldots, atom_m\}$.

Query atoms can be *concept* query atoms ($C(X)$), *role* query atoms ($R(X, Y)$), *same-as* query atoms ($X = Y$) as well as so-called *concrete domain* query atoms. The latter are introduced to provide support for querying the concrete domain part of a knowledge base and will not be covered in detail here. Complex queries are built from query atoms using boolean constructs for conjunction (indicated with comma) or union ($\vee$).

In *standard* conjunctive queries, variables (in the head and in query atoms in the body) are bound to (possibly anonymous) domain objects. A system supporting (unions of) standard conjunctive queries is QuOnto. In so-called *grounded* conjunctive queries, $C(X)$, $R(X, Y)$ or $X = Y$ are true if, given some bindings α for mapping from variables to *individuals mentioned in the Abox* $\mathcal{A}$, it holds that $(\mathcal{T}, \mathcal{A}) \models \alpha(X) : C$, $(\mathcal{T}, \mathcal{A}) \models (\alpha(X), \alpha(Y)) : R$, or $(\mathcal{T}, \mathcal{A}) \models \alpha(X) = \alpha(Y)$, respectively. In grounded conjunctive queries the standard semantics can be obtained for so-called tree-shaped queries by using corresponding existential restrictions in query atoms.

In practical applications it is advantageous to name subqueries for later reuse, and practical systems, such as for instance RacerPro, support this for grounded conjunctive queries with non-recursive rules of the following form

$$P(X_1, \ldots, X_{n_1}) \leftarrow A_1(Y_1), \ldots, A_l(Y_l), R_1(Z_1, Z_2), \ldots, R_h(Z_{2h-1}, Z_{2h}). \tag{2.1}$$

The predicate term to the left of $\leftarrow$ is called the head and the rest is called the body (a set of atoms), which, informally speaking, is seen as a conjunction of predicate terms. All variables in the head have to occur in the body, and rules have to be non-recursive (with the obvious definition of non-recursivity). Since rules have to be non-recursive, the replacement of query atoms matching a rule head is possible (unfolding, with the obvious definition of matching). The rule body is inserted (with well-known variable substitutions and variable renamings). If there are multiple rules (definitions) for the same predicate P, corresponding disjunctions are generated. The unfolding process starts with the set of atoms of a query. Thus, we start with a set of atom sets.

$$\{\{atom_1, atom_2, \ldots atom_k\}\}$$

Each element of the outer set represents a disjunct. Now, wlog we assume that there are n rules matching $atom_2$. Then, the set $\{atom_1, atom_2, \ldots atom_k\}$ is eliminated and replaced with the sequence of sets $\{atom_1\} \cup$ replace_vars(body($rule_1$), $head(rule_1), atom_2) \cup \{\ldots atom_k\}$, $\ldots, \{atom_1\} \cup$ replace_vars(body $(rule_n), head(rule_n), atom_2) \cup \{\ldots atom_k\}$. The unfolding process proceeds until no replacement is possible any more (no rules match). The unfold operator is used in the abduction process, which is described in the next section.

2.3 Abduction as a Non-Standard Inference Service

In this report, we argue that abduction can be considered as a new type of non-standard retrieval inference service. In this view, observations (or part of them) are utilized to constitute queries that have to be answered. Contrary to existing retrieval inference services, answers to a given query cannot be found by simply exploiting the knowledge base. In fact, the abductive retrieval inference service has the task of acquiring what should be added to the knowledge base in order to positively answer a query.

More formally, for a given set of Abox assertions Γ (in form of a query) and a knowledge base $\Sigma = (\mathcal{T}, \mathcal{A})$, the abductive retrieval inference service aims to derive all sets of Abox assertions Δ (explanations) such that $\Sigma \cup \Delta \models \Gamma$ and the following conditions are satisfied:

- $\Sigma \cup \Delta$ is satisfiable, and
- Δ is a minimal explanation for Γ, i.e., there exists no other explanation Δ' in the solution set that is not equivalent to Δ and it holds that $\Sigma \cup \Delta' \models \Delta$.

In addition to minimality (simplicity), in [16] another dimension called consilience is mentioned. An explanation should explain as many elements of Γ as possible. Both measures are contradictory.

In the next section, we will focus on the use of abductive retrieval inference services for multimedia interpretation and address two important issues, namely finding explanations that meet the conditions listed above and selecting 'preferred' ones.

2.4 Interpretation of Multimedia Documents

For intelligent retrieval of multimedia documents such as images, videos, audio, and texts, information extracted by media analysis techniques has to be enriched by applying high-level interpretation techniques. The interpretation of multimedia content can be defined as the recognition of abstract knowledge, in terms of concepts and relations, which are not directly extractable by low-level analysis processes, but rather require additional high-level knowledge. Furthermore, such abstract concepts are represented in the background knowledge as aggregate concepts with constraints among its parts.

In this section, we start by specifying the requirements for the abduction approach by defining its input and output. Then, we proceed with describing the framework for generating explanations, and finally introduce a scenario with a particular example involving for image interpretation where various explanations are generated and the usefulness of a preference score is demonstrated.

2.4.1 Requirements for Abduction

The abduction approach requires as input a knowledge base Σ consisting of a Tbox $\mathcal{T}$ and an Abox $\mathcal{A}$. We assume that the information extracted from a multimedia document through low-level analysis (e.g., image analysis) is formally encoded as a set of Abox assertions (Γ). For example, in the context of images for every object recognized in an image, a corresponding concept assertion is found in Γ. Usually, the relations that can be extracted from an image are spatial relations holding among the objects in the image. These relations are also represented as role assertions in Γ. In order to construct a high-level interpretation of the content in Γ, the abduction process will extend the Abox with new concept and role assertions describing the content of the multimedia document at a higher level.

The output of the abduction process is formally defined as a set of assertions Δ such that $\Sigma \cup \Delta \models \Gamma$, where $\Sigma = (\mathcal{T}, \mathcal{A})$ is the knowledge base (usually the Abox A is assumed to be empty), Γ is a given set of low-level assertions, and Δ is an explanation, which should be computed. The solution Δ must satisfy certain side conditions (see Section 2.3). To compute the explanation Δ in our context we modify this equation into

$$\Sigma \cup \Gamma_1 \cup \Delta \models \Gamma_2, \tag{2.2}$$

where the assertions in Γ will be split into bona fide assertions (Γ_1) and assertions requiring fiats (Γ_2).[1] Bona fide assertions are assumed to be true by default, whereas fiat assertions are aimed to be explained. The abduction process tries to find explanations (Δ) such that Γ_2 is entailed. This entailment decision is implemented as (boolean) query answering. The output Δ of the abduction process is represented as an Abox. Multiple solutions are possible.

2.4.2 The Abduction Framework

The abduction framework exploits the non-recursive rules of Σ to answer a given query in a backward-chaining way (see Framework 1). The function `compute_explanations` gets Σ, Γ_1, and Γ_2 as input. We assume a function `transform_into_query` that is applied to a set of Abox assertions Γ_2 and returns a set of corresponding query atoms. The definition is obvious and left out for brevity. Since the rules in Σ are non-recursive, the unfolding step (see Line 2 in Framework 1) in which each atom in the transformed Γ_2 is replaced by the body of a corresponding rule is well-defined. The function `unfold` returns a set of atom sets (each representing a disjunct introduced by multiple matching rules, see above).

The function `explain` computes an explanation Δ for each $\gamma \in \Gamma_2'$. The function `vars` (or `inds`) returns the set of all vars (or inds) mentioned in the argument structures. For each variable in γ a new individual is generated (see the set *new_inds* in Line 7). Besides old individuals, these new individuals are used in a non-deterministic variable substitution. The variable substitution σ_{γ,new_inds} (line 8) is inductively extended as follows:

- $\sigma_{\gamma,new\_inds}(\{a_1, \ldots, a_n\}) =_{def} \{\sigma_{\gamma,new\_inds}(a_1), \ldots, \sigma_{\gamma,new\_inds}(a_n)\}$
- $\sigma_{\gamma,new\_inds}(C(x)) =_{def} C(\sigma_{\gamma,new\_inds}(x))$
- $\sigma_{\gamma,new\_inds}(R(x,y)) =_{def} R(\sigma_{\gamma,new\_inds}(x), \sigma_{\gamma,new\_inds}(y))$
- $\sigma_{\gamma,new\_inds}(x) =_{def} x$ if x is an individual

[1] With the obvious semantics we slightly abuse notation and allow a tuple of sets of assertions Σ to be unioned with a set of assertions $\Gamma_1 \cup \Delta$.

The function `transform` maps $C(i)$ into $i : C$ and $R(i, j)$ into $(i, j) : R$, respectively. All satisfiable explanations Δ derived by `explain` are added to the set of explanations Δs. The function `compute-preferred-explanations` transforms the Δs into a poset according to a preference measure and returns the maxima as a set of Aboxes. The preference score of a Δ used for the poset order relation is: $S_{pref}(\Delta) := S_i(\Delta) - S_h(\Delta)$ where S_i and S_h are defined as follows.

- $S_i(\Delta) := |\{i | i \in inds(\Delta) \text{ and } i \in inds(\Sigma \cup \Gamma_1)\}|$
- $S_h(\Delta) := |\{i | i \in inds(\Delta) \text{ and } i \in new\_inds\}|$

Algorithm 1 The Abduction Framework

1: **function** compute_explanations$(\Sigma, \Gamma_1, \Gamma_2, S)$: set of Aboxes
2: $\Gamma_2' :=$ unfold(transform_into_query$(\Gamma_2), \Sigma)$ // $\Gamma_2' = \{\{atom_1, \ldots, atom_m\}, \ldots\}$
3: $\Delta s := \{\Delta \mid \exists \gamma \in \Gamma_2'.(\Delta =$ explain$(\Sigma, \Gamma_1, \gamma), \Sigma \cup \Gamma_1 \cup \Delta \not\models \bot)\}$
4: **return** compute_preferred_explanations$(\Sigma, \Gamma_1, \Delta s, S)$

5: **function** explain$(\Sigma, \Gamma_1, \gamma)$: Abox
6: $n := |\text{vars}(\gamma)|$
7: $new\_inds := \{new\_ind_i \mid i \in \{1 \ldots n\}\}$, where $new\_inds \cap (inds(\Sigma) \cup inds(\Gamma_1)) = \emptyset$
8: $\Delta := \{\text{transform}(a) \mid \exists \sigma_{\gamma, new\_inds} : vars(\gamma) \mapsto (inds(\Sigma) \cup inds(\Gamma_1) \cup new\_inds).$
9: $(a \in \sigma_{\gamma, new\_inds}(\gamma), (\Sigma \cup \Gamma_1) \not\models a)\}$
10: **return** Δ

11: **function** compute_preferred_explanations$(\Sigma, \Gamma_1, \Delta s, S)$: set of Aboxes
12: **return** maxima(poset$(\Delta s, \lambda(x, y) \bullet S(x) < S(y))$)

Depending on the preference function given as the actual parameter for the argument S, the procedure `compute_explanations` can be considered as an approximation w.r.t. the minimality and consilience condition defined in Section 2.3. It adds to the explanation those query atoms that cannot be proven to hold.

For the abduction framework, only the rules are considered. The GCIs should be used for abduction as well, however. We might accomplish this by approximating the Tbox with the DLP fragment and, thereby, see the Tbox axioms from a rules perspective in order to better reflect the Tbox in the abduction process. The procedure does not add irrelevant atoms (spurious elements of an explanation), in case the rules are well-engineered and do not contain irrelevant ways to derive assertions. The procedure could be slightly modified to check for those redundancies.

2.4.3 An Example for Image Interpretation as Abduction

For the image shown in Figure 2.1, we suppose the Abox in Figure 2.2 is provided by low-level image analysis. Furthermore, a sample Tbox of the athletics domain and a small set of rules are assumed to be provided as background knowledge Σ (see Figure 2.3). In order to keep the example as brief as possible, only a very small fragment of the ontology developed in work package 3 of the Boemie project is used here.

In order to find a 'good' high-level interpretation of this image, we divide the Abox Γ into Γ_1 and Γ_2 following Equation 2.2. In this example Γ_1 contains $\{pole_1 : Pole, human_1 : Human, bar_1 : Bar\}$ and Γ_2 contains $\{(bar_1, human_1) : near\}$. Consequently, the abductive retrieval inference service computes the following boolean query in line 2: $Q_1 := \{() \mid near(bar_1, human_1)\}$. In this report we do not elaborate on the strategy to determine which Γ_2 to actually choose.

Figure 2.1: A pole vault event.

$$
\begin{array}{rcl}
pole_1 & : & Pole \\
human_1 & : & Human \\
bar_1 & : & Bar \\
(bar_1,\, human_1) & : & near
\end{array}
$$

Figure 2.2: An Abox Γ representing the results of low-level image analysis.

$$
\begin{array}{rcl}
Jumper & \sqsubseteq & Human \\
Pole & \sqsubseteq & Sports\_Equipment \\
Bar & \sqsubseteq & Sports\_Equipment \\
Pole \sqcap Bar & \sqsubseteq & \bot \\
Pole \sqcap Jumper & \sqsubseteq & \bot \\
Jumper \sqcap Bar & \sqsubseteq & \bot \\
Jumping\_Event & \sqsubseteq & \exists_{\leq 1} hasParticipant.Jumper \\
Pole\_Vault & \sqsubseteq & Jumping\_Event \sqcap \exists hasPart.Pole \sqcap \exists hasPart.Bar \\
High\_Jump & \sqsubseteq & Jumping\_Event \sqcap \exists hasPart.Bar \\
near(Y,Z) & \leftarrow & Pole\_Vault(X), hasPart(X,Y), Bar(Y), \\
 & & hasPart(X,W), Pole(W), hasParticipant(X,Z), Jumper(Z) \\
near(Y,Z) & \leftarrow & High\_Jump(X), hasPart(X,Y), Bar(Y), \\
 & & hasParticipant(X,Z), Jumper(Z)
\end{array}
$$

Figure 2.3: A tiny example Σ consisting of a Tbox and DL-safe rules.

Obviously, both rules in Σ match with the 'near' atom in query Q_1. Therefore, the abduction framework first generates explanations by non-deterministically substituting variables in the query body with different instances from Γ_1 or with new individuals. Some intermediate Δ results turn out to be unsatisfiable (e.g., if the bar is made into a pole by the variable subsitution process). However, several explanations still remain as possible interpretations of the image. The preference score is used to identify the 'preferred' explanations. For example, considering the following explanations of the image

- $\Delta_1 = \{new\_ind_1 : Pole\_Vault,\ (new\_ind_1, bar_1) : hasPart,\ (new\_ind_1, new\_ind_2) : hasPart,\ new\_ind_2 : Pole,\ (new\_ind_1, human_1) : hasParticipant,\ human_1 : Jumper\}$
- $\Delta_2 = \{new\_ind_1 : Pole\_Vault,\ (new\_ind_1, bar_1) : hasPart,\ (new\_ind_1, pole_1) : hasPart,\ (new\_ind_1, human_1) : hasParticipant,\ human_1 : Jumper\}$
- $\Delta_3 = \{new\_ind_1 : High\_Jump,\ (new\_ind_1, bar_1) : hasPart,\ (new\_ind_1, human_1) : hasParticipant,\ human_1 : Jumper\}$

the preference measure of Δ_1 is calculated as follows: Δ_1 incorporates the individuals $human_1$ and bar_1 from Γ_1 and therefore $S_i(\Delta_1)$=2. Furthermore, it hypothesizes two new individuals, namely new_ind_1 and new_ind_2, such that $S_h(\Delta_1)$=2. The preference score of Δ_1 is $S(\Delta_1)$= $S_i(\Delta_1)$- $S_h(\Delta_1)$=0. Similarly, the preference scores of the second and third explanations are $S(\Delta_2)$=2 and $S(\Delta_3)$=1. After transforming the Δs into a poset, the algorithm computes the maxima. In our case, the resulting set of Aboxes contains only one element, Δ_2, which represents the 'preferred' explanation. Indeed, the result is plausible, since this image should better be interpreted as showing a pole vault and not a high jump, due to the fact that low-level image analysis could detect a pole, which should not be ignored as in the high-jump explanation.

2.5 Evaluation

The overall goal of the framework is to provide high-level content descriptions of media documents for maximizing precision and recall of semantics-based information retrieval. In this section, we provide an empirical evaluation of the results of the framework on a collection of athletics images in order to analyze the utility of the framework.

For this purpose, we implemented the media interpretation framework discussed above. The core component of this implementation is the DL-reasoner RacerPro that supports various inference services. The abductive retrieval inference service, which is the key inference service for media interpretation, is integrated into the latest version of RacerPro. The framework gets analysis Aboxes, exploits various inference services of RacerPro, and returns interpretation Aboxes as high-level content descriptions. For the time being, the computation of preference scores is not implemented and, therefore, interpretation Aboxes contain all possible explanations.

To test the implementation, we used an ontology about the athletics domain and an image corpus. The corpus consists of images showing either a pole vault or a high jump event. The images have been manually annotated with annotation tools in order to train low-level feature extractors for prospective athletics corpora. I.e., using the annotation tools, annotators manually annotated regions of images (as visual representations of concepts), with corresponding concepts from the ontology such as *Pole*, *Bar* and *PersonFace*. Afterwards, annotated images have been analyzed automatically to detect relations between concept instances. Finally, for each image in the corpus an analysis Abox with corresponding assertions has been generated.

We tested the implementation in the following setup: the aggregate concepts *PoleVault* and *HighJump* from the domain ontology are defined as target concepts. Analysis Aboxes of pole vault and high jump images are used as input for high-level media interpretation. The results obtained for pole vault and high jump images are shown in Figure 2.4 and 2.5, respectively. To analyze the usefulness of the results for information retrieval, in both figures interpretation Aboxes are categorized w.r.t. the existence (or absence) of aggregate concept instances: *A*) contains no aggregate concept instances at all *B*) contains an aggregate concept instance but no target concept instance *C*) contains a *HighJump* and a *PoleVault* instance *D*) contains a *PoleVault* instance *E*) contains more than one *PoleVault* instances and one or no *HighJump* instances

At first sight, only interpretation Aboxes that fall into the category *D* in Figure 2.4 look like 'good' interpretation results for pole vault images, because the corresponding images are annotated with a single *PoleVault* instance. However, if the implementation would be enhanced to include preference scores, as discussed in Section 2.4.2 for an example pole vault image, all interpretation Aboxes of category *C* and *E* would include the most 'preferred' explanation only (in this case a single *PoleVault* instance), and hence fall into the category *D*, too.

Both in Figure 2.4 and 2.5, category *A* interpretation Aboxes are identical to the corresponding analysis Aboxes and indicate that no new knowledge could be inferred through high-level interpretation. For other images (category *B* interpretation Aboxes) high-level interpretation infers new knowledge (including an aggregate concept instance) but fails to derive an instance of the target concepts.

In fact, category *B* interpretation Aboxes contain a *Person* instance to explain the existence of *PersonBody* and *PersonFace* instances and their constellation in the image. Deeper analysis of category *A* and *B* interpretation Aboxes showed that insufficient interpretation results are caused by the failure of image analysis to extract some of the existing relations in the corresponding images. Taking into account the ambiguity and uncertainty involved in the image analysis

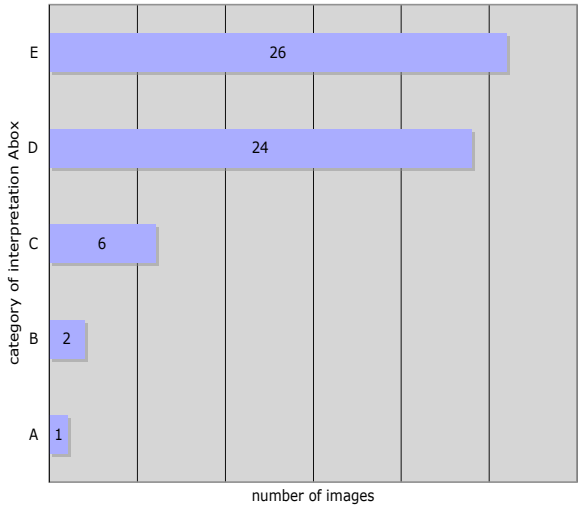

Figure 2.4: Results for pole vault images.

process, this information (the failure of adequate interpretation) can be used to create a valuable feedback for the image analysis tools.

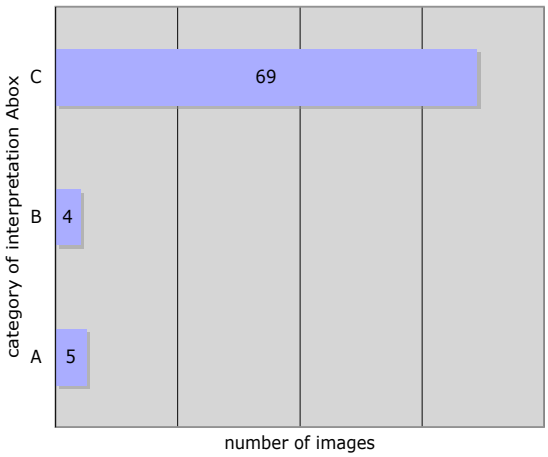

Figure 2.5: Results for high jump images.

Figure 2.5 shows that every high jump image is interpreted as either showing a high jump or a pole vault event (category C), besides incompletely analyzed ones, which fall into the categories *A* or *B*. Different from pole vault images, interpretations of high jump images cannot be disambiguated through preference scores. This result indicates that necessary rules are missing in the background knowledge due to the fact that, currently, image analysis cannot extract distinctive features of high jump images. Note that appropriate fusion of information from different modalities can help to solve this problem, even if image analysis results cannot be improved. For example, if an image is captioned with text, text analysis can extract additional information that can be used to disambiguate the interpretation of the image.

Our experiments showed that, if provided with an appropriate ontology and low-level annotations, the existing implementation of the media interpretation framework delivers promising results for images and can be used for maximizing precision and recall of semantics-based information retrieval systems.

2.6 Summary

In this chapter we presented a novel approach to interpret multimedia data using abduction with description logics that makes use of a new type of non-standard retrieval service in DLs. We showed that results from low-level media analysis can be enriched with high-level descriptions using our Abox abduction approach. In this approach, backward-chained DL-safe-like rules are exploited for generating explanations. For each explanation, a preference score is calculated in order to implement the selection of 'preferred' explanations. Details of the approach have been discussed with a particular example for image interpretation.

The idea of formalizing interpretation as abduction is investigated in [16] in the context of text interpretation. In [38], Shanahan presents a formal theory of robot perception as a form of abduction, where low-level sensor data is transformed into a symbolic representation of the world in first-order logic and abduction is used to derive explanations. In [9] a detailed discussion of abductive reasoning tasks in DLs including Abox abduction is presented. The authors consider the development of algorithmic techniques based on semantic tableaux for employing abductive inference in expressive DLs as the most promising approach. However, a solution to the Abox abduction problem is formally presented, but for the time being it is not shown how to derive solutions.

The abduction approach we follow in this report is based on the combination of the works in [16], [38] and [29]. In contrast to approaches such as [18], which use abduction in the context of rules in logic programming only, we combine existing DL reasoning mechanisms and rules in a coherent framework and consider abduction as a new type of non-standard retrieval inference service, which is integrated into existing DL reasoners.

In this report we presented a media interpretation framework that leverages low-level information extraction to a higher level of abstraction and, therefore, enables the automatic annotation of documents through high-level content descriptions. The availability of high-level content descriptions for media documents will enable semantics-based information retrieval using more abstract terms, which is essential for the Semantic Web. The key inference service used by this framework is the abductive retrieval inference service that generates explanations for observations.

The empirical evaluation presented in this report indicates that a coherent framework incorporating appropriate ontology design, dedicated low-lewel IE and reasoning in DLs delivers promising results for media interpretation. Further analysis of test results showed that the implementation of the proposed preference score will enhance the results of media interpretation and, therefore, contribute to the maximization of precision and recall of semantics-based information retrieval.

Currently, we are investigating fusion of information from different modalities to enhance media interpretation results. In our future work, we will investigate inductive learning of DL-safe rules for abduction using training data.

Chapter 3

Event Recognition Facility

In some applications it might be interesting to state that assertions are valid only within a certain time interval. This is not possible with standard description logic systems. There are several research results available on qualitative temporal reasoning in the context of description logics. RacerPro can support qualitative temporal reasonsing as part of the nRQL system (compare the section on RCC substrates in the User's Guide). In addition, in some cases, quantitative information about time intervals might be available and could be relevant for query answering. RacerPro now also supports Abox query answering w.r.t. assertions that are associated with specifications for time intervals.

Algorithms for answering queries involving rules with time variables have been published in [31] and [30]. In addition to the original Prolog-style approach in [30], conjunctive query atoms for Aboxes are provided for queries with time variables in RacerPro.

3.1 Temporal Propositions:

We assume that assertions involving time variables such as, e.g. "ind_1 approaching ind_2 from t_1 to t_2" are generated by low-level image sequence analysis processes. The results are added to an ABox as so-called temporal propositions.

A *temporal proposition* is a syntactic structure of the following form:

$$P_{[t_1,t_2]}(ind_1, \ldots, ind_n)$$

where t_i denotes an element of a linear temporal structure $\Theta \subseteq \mathbb{N}$, ind_i with $i \in \{1, \ldots, n\}$ denotes an individual, and $P \in Preds$.

The semantics for rules with time intervals is different from DL-safe rules, and formally defined as follows. Let $\Theta \subseteq \mathbb{N}$ be a linear temporal structure. A temporal interpretation $\mathcal{I}_T$ is a tuple $(\Delta, \cdot^{\mathcal{I}}, \Theta, \Im)$ such that, in addition to the standard components of an interpretation, $\Im$ is an injective mapping from the temporal structure Θ to a set of standard Tarskian interpretation functions as used in previous sections.

A temporal interpretation $\mathcal{I}_T = (\Delta, \cdot^{\mathcal{I}}, \Theta, \Im,)$ satisfies a GCI or an ABox assertion if the standard part $(\Delta, \cdot^{\mathcal{I}})$ satisfies the GCI or the ABox assertion. The remaining components are used for defining satisfiability of temporal propositions. A temporal interpretation $\mathcal{I}_T$ satisfies a temporal proposition $P_{[t_1,t_2]}(ind_1, \ldots, ind_n)$ if the predicate is true for all time points in the non-empty interval $[t_1, t_2]$. Hence, we assume that temporal propositions are durative, i.e., the proposition

holds for all non-empty subintervals for a more detailed analysis. More formally:

$$\mathcal{I}_T \models P_{[t_1,t_2]}(ind_1, \ldots ind_n)$$

if for all $\theta \in \Theta$, $|\Theta| > 1$, it holds that if $t_1 \leq \theta \leq t_2$, then $(ind_1^{\Im(\theta)}, \ldots, ind_n^{\Im(\theta)}) \in P^{\Im(\theta)}$. As usual, a temporal interpretation that satisfies a temporal proposition is called a *temporal model* for this term. A temporal interpretation which satisfies a GCI or an ABox assertion is called a (temporal) model for the GCI or ABox assertion, respectively. An Abox with a set of temporal propositions such as, e.g. $\{move\_forward_{[10,20]}(ind_1), move\_backward_{[10,20]}(ind_1)\}$ should be inconsistent, but this requires (TBox) knowledge about the disjointness of predicates $move\_forward$ and $move\_backward$ for all time points. We ignore these issues here.

3.2 Queries w.r.t. Temporal Propositions:

In order to support event recognition in an ontology-based media interpretation system, we introduced temporal propositions. For queries over ABoxes that also contain temporal propositions, rules with time intervals can be defined. Suppose three disjoint sets of names $Preds$, $TimeVars$ and $Vars$ neither of which is a subset of the names mentioned in the axioms of the ontology. Then, a *rule with time intervals* has the following structure:

$$\begin{array}{lll} P_{[T_0,T_1]}(X_1, \ldots, X_{n_1}) & \leftarrow & Q_{1[T_2,T_3]}(Y_{1,1}, \ldots, Y_{1,m_1}), \\ & & \ldots \\ & & Q_{k[T_{2k},T_{2k+1}]}(Y_{k,1}, \ldots, Y_{k,m_k}), \\ & & A_1(Z_1), \\ & & \ldots \\ & & A_l(Z_l), \\ & & R_1(W_1, W2), \\ & & \ldots \\ & & R_h(W_{2h-1}, W_{2h}). \end{array}$$

where the $T_i \in TimeVars$ are temporal variables and $X_i, Y_{j,k}, Z_l, W_h \in Vars$ are (not necessarily disjoint) variables that are bound to individuals mentioned in the ABox, $P, Q_i \in Preds$, and A_j, R_k are concept names and role names, respectively. In a similar way as for conjunctive queries introduced above, all variables in the head must be mentioned in the body, and rules must be non-recursive. Thus, queries w.r.t. time variables are unfolded, similar to rules for defined queries.

A *conjunctive query with time variables* is an expression of the following form:

$$\begin{array}{lll} \{(X_1, \ldots, X_n)_{[T_1,T_2]} & | & Q_{1[T_2,T_3]}(Y_{1,1}, \ldots, Y_{1,m_1}), \\ & & \ldots \\ & & Q_{k[T_{2k},T_{2k+1}]}(Y_{k,1}, \ldots, Y_{k,m_k}), \\ & & A_1(Z_1), \\ & & \ldots \\ & & A_l(Z_l), \\ & & R_1(W_1, W2), \\ & & \ldots \\ & & R_h(W_{2h-1}, W_{2h})\} \end{array}$$

Note that the variables $X_i, Y_{i,j}, Z_i$, and W_i are not necessarily disjoint. A tuple $(ind_1, \ldots, ind_n)_{[t_1,t_2]}$ is a potential solution of a grounded unfolded temporal conjunctive query (temporal query for short) if the variable substitution $[X_1 \leftarrow ind_1, \ldots, X_n \leftarrow ind_n, T_1 \leftarrow t_1, T_2 \leftarrow t_2]$ can

be extended with additional assignments for all other variables in the body such that the resulting query atoms after applying the substitution are satisfied in all temporal models of the ontology (TBox and ABox). The result set for a temporal query comprises all tuples $(ind_1, \ldots, ind_n)_{[(t_{1_{min}}, t_{1_{max}}),(t_{2_{min}}, t_{2_{max}})]}$ such that there exists no other potential solution $(ind_1, \ldots, ind_n)_{[t_1,t_2]}$ with $t_1 < t_{1_{min}}$ or $t_1 > t_{1_{max}}$ or $t_2 < t_{2_{min}}$ or $t_2 > t_{2_{max}}$.

3.3 Example: Detecting High-Jump Events

Time intervals are important for recognizing events in image sequences. In contrast to still images, events in image sequences have a temporal extension that has to be appropriately considered for constructing media interpretations. In order to detect high-level events such as "high-jump", event predicates are described using rules with time variables. For high-jump events we sketch the rule design pattern in Figure 3.1. In our approach we suppose that basic events can be detected by image analysis processes. Basic events are described with temporal propositions (being added to an interpretation ABox by low-level processes). An example is shown in Figure 3.2.

$$
\begin{aligned}
High\_Jump\_Event_{[T_1,T_2]}(X,Y) \quad \leftarrow \quad & accelerate\_horizontally_{[T_1,T_3]}(Y), \\
& vertical\_upward\_movement_{[T_3,T_4]}(Y), \\
& turn_{[T_4,T_5]}(Y), \\
& vertical\_downward\_movement_{[T_5,T_2]}(Y). \\
& Jumper(Y), \\
& High\_Jump(X), \\
& hasPart(X,Y),
\end{aligned}
$$

Figure 3.1: Rule with time intervals for recognizing high jump events.

$$
\begin{aligned}
accelerate\_horizontally_{[219,224]} \quad & (moving\_object_1) \\
vertical\_upward\_movement_{[224,226]} \quad & (moving\_object_1) \\
turn_{[226,228]} \quad & (moving\_object_1) \\
vertical\_downward\_movement_{[228,230]} \quad & (moving\_object_1) \\
moving\_object_1 : \quad & Jumper \\
event_1 : \quad & High\_Jump \\
(event_1, moving\_object_1) : \quad & hasPart
\end{aligned}
$$

Figure 3.2: Abox assertions for basic events (temporal propositions) that are detected by image sequence analysis components. In addition, three standard assertions possibly extracted from other sources (e.g. images and text) are added.

In order to actually recognize events for particular individuals which satisfy restrictions w.r.t. the ontology, the query language for temporal propositions introduced in Section 2.2 is applied. An example for a query involving events and time intervals is shown below.

$$\{(X)_{[T_1,T_2]} \mid High\_Jump\_Event_{[T_1,T_2]}(X,Y)\}$$

To answer a query, two steps have to be carried out. First, an assignment α for query variables (i.e. X in the query shown above) has to be found such that the body predicate terms and atoms are satisfied. Second, the goal is to determine lower bound and upper bound values for the temporal variables (T_1, T_2 in the example) such that the temporal propositions in the query body are satisfied. The result of the example query is $(event_1)_{[(219,223),(229,230)]}$. Thus, for all

$T_1 \in (219, 223)$ and $T_2 \in (229, 230)$ and for all remaining temporal variables in the body of the rule in Figure 3.1 there exist values such that all predicate terms in the body are satisfied with the assignment $\alpha(X) \rightarrow event_1$.

If a high-jump event is expected but the query for the high-jump event (see above) returns $false$, then abduction can be used to determine what has to be added to the interpretation ABox.

3.4 Example in Racer Syntax

The following example shows a concrete example using the syntax supported by RacerPro (the example is inspired by the traffic-scene domain used in the original work published in [31] and [30]).

```
(in-knowledge-base traffic-analysis)

(define-primitive-role r :inverse r)

(implies car vehicle)
(implies volkswagen car)

(instance vw1 volkswagen)
(instance vw2 volkswagen)
(instance ralf pedestrian)
(related vw2 vw1 r)

(define-event-assertion ((move vw1) 7 80))
(define-event-assertion ((move vw2) 3 70))
(define-event-assertion ((move ralf) 3 70))
(define-event-assertion ((approach vw1 vw2) 10 30))
(define-event-assertion ((behind vw1 vw2) 10 30))
(define-event-assertion ((beside vw1 vw2) 30 40))
(define-event-assertion ((in-front-of vw1 vw2) 40 80))
(define-event-assertion ((recede vw1 vw2) 40 60))

(define-event-rule ((overtake ?obj1 ?obj2) ?t1 ?t2)
  ((?obj1 car) ?t0 ?tn)
  ((?obj1 ?obj2 r) ?t0 ?tn)
  ((move ?obj1) ?t0 ?t2)
  ((move ?obj2) ?t1 ?t2)
  ((approach ?obj1 ?obj2) ?t1 ?t3)
  ((behind ?obj1 ?obj2) ?t1 ?t3)
  ((beside ?obj1 ?obj2) ?t3 ?t4)
  ((in-front-of ?obj1 ?obj2) ?t4 ?t2)
  ((recede ?obj1 ?obj2) ?t4 ?t2))
```

The following query checks whether there exists an overtake event hidden in the Abox and the event assertions.

```
(timenet-retrieve ((overtake ?obj1 ?obj2) ?t1 ?t2))
```

The query returns a set of binding for variables if an event can be detected (and nil otherwise). For time variables an interval for the lower-bound and upper-bound are returned. Thus, the `overtake` event start at time unit 10 at the earliest and 29 at the latest. It ends at time unit 29 at the earliest and 60 at the latest. In a future version of RacerPro, redundant binding specifications will be removed.

```
(((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60))))
```

Note that the example involves reasoning. The fact that `vw1` is a car is only implicitly stated. In addition, temporal constraints have to be checked for the time intervals.

In the previous queries, variables are used. However, one might very well use constants in event queries as shown in the following example.

```
(timenet-retrieve ((overtake ?obj1 vw2) ?t1 ?t2))
```

Now, only bindings for the variable `?obj1` are returned.

```
(((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60))))
```

In a future version, a forward-chaining application of event rules will be supported, so there is no need to repeatedly cycle through all known events using respective queries.

Chapter 4

Nonstandard Inferences

RacerPro 1.9.1 provides a prototypical implementation of the LCS operator (least-common subsumer) for the description logic $\mathcal{ALE}$ with unfoldable Tboxes. Furthermore, the MSC-k (most-specific concept up to k levels of nested existentials) is supported. In order to demonstrate the application of these operators, the following knowledge base is used.

4.1 Least-Common Subsumer

The least common subsumer of a given collection of concept descriptions is a description that represents the properties that all the elements of the collection have in common. More formally, it is the most-specific concept description that subsumes the given descriptions:

Definition: The least common subsumer of a given collection of concept descriptions is a description that represents the properties that all the elements of the collection have in common. More formally, it is the most-specific concept description that subsumes the given descriptions:

Definition: Let $\mathcal{L}$ be a description language. A concept description E of $\mathcal{L}$ is the least common subsumer (lcs) of the concept descriptions $C_1, \ldots, C_n$ in $\mathcal{L}$ ($lcs(C_1, \ldots, C_n)$ for short) iff it satisfies

1. $C_i \sqsubseteq E$ for all $i = 1, \ldots, n$, and
2. E is the least $\mathcal{L}$-concept description satisfying (1.), i.e., if E_0 is a $\mathcal{L}$-concept description satisfying $C_i \sqsubseteq E_0$ for all $i = 1, \ldots, n$, then $E \sqsubseteq E_0$. As an easy consequence of this definition, the lcs is unique up to equivalence.

4.2 Most-Specific Concept up to Nesting Level k

The most-specific concept is defined as follows:

Definition: A concept description E in some description language $\mathcal{L}$ is the most-specific concept (MSC) of a individual a defined in an ABox A ($msc(a)$ for short) iff

1. $A \models E(a)$, and
2. E is the most-specific concept satisfying (1.), i.e., if E_0 is an $\mathcal{L}$ -concept description satisfying $A \models E_0(a_i)$ for all $i = 1, \ldots, n$, then $E \sqsubseteq E_0$.

The description language we consider here is $\mathcal{ALE}$. Usually, no finite most-specifc concept exists even for simple Aboxes. Therefore, we consider the most-specific concept up to a maximal nesting level k of existential restrictions (MSC-k).

4.3 Example

```
(in-knowledge-base test)

(equivalent x (some r (and a d)))
(equivalent y (some r b))
(implies a b)
(implies c b)

(instance j a)
(instance i (all r (and (some r b) d)))
(related i j r)
(related j k r)
(related k i r)
```

We now apply the LCS operator to two input concepts.

```
(lcs-unfold x (and (all r d) (some r c)))
```

The result is `(some r (and d b))`. If no unfolding is desired, use `lcs` instead of `lcs-unfold`. It is possible to specify the Tbox as an optional last argument (default is the current Tbox).

For some purposes, an extract of the information contained in an Abox might be useful. RacerPro 1.9.1 provides an implementation of the MSC-k operator. MSC-k is applied to an individual and a nesting depth. For instance, given the knowledge base above, the following form is executed.

```
(msc-k i 3)
```

The result is

```
(and (some r (and a
                  (some r (and (some r (and (all r (and (some r b) d))
                                            (some r (and a top)))))))))
```

It is possible to supply an additional boolean argument which indicates whether for each individual, the direct types are included in the MSC approximation. For instance,

```
(msc-k i 3 t)
```

returns

```
(and (some r
           (and x
```

```
a
(some r
      (and a
           d
           y
           (some r (and *top*
                        (all r (and (some r b) d))
                        (some r (and x a top)))))))))
```

The default for the third argument is `nil` (indicating that direct types are not to be included in the MSC-k output). Given this boolean argument is specified, there can be a fourth argument, the Abox (which defaults to the current Abox).

Chapter 5

Extensions to nRQL

In this section, we describe the extension of nRQL by means of a simple expression language called "MiniLisp". With MiniLisp a user can write a simple, termination safe, "program" which is executed on the RacerPro server. Such MiniLisp programs can be used to improve the flexibility of the query answering. MiniLisp allows a restricted kind of server-side programming and can thus be used to implement, for example

1. user-defined output formats for query results (e.g., the query results can also be written into a file),
2. certain kinds of combined ABox/TBox queries,
3. efficient aggregation operators (e.g., sum, avg, like in SQL),
4. user-defined projection operators.

The functional lambda expressions in MiniLisp can be used in query heads and rule antecedences.

The MiniLisp extension discussed in this section is presented in the latest RacerPro users guide.

5.1 Lambda Head Operators to Evaluate Expressions

A so-called head projection operator is a function which is applied to the current binding of the variable (the current individual), and the operator result is included in the answer tuple. The application of a head projection operator can be understood as a function application. So-called *lambda expressions* can denote (anonymous) functions. nRQL allows for the specification of lambda expressions in the head of a query; nRQL uses a Lisp dialect which we call MiniLisp. MiniLisp is a termination-safe expression language (not a programming language).

For example, consider an ABox representing material objects in the physical world, having width and length, and we want to computed and return the area of these objects with a query:

```
? (define-concrete-domain-attribute width :type integer)
> :OKAY

? (define-concrete-domain-attribute length :type integer)
> :OKAY
```

```
? (instance i (and (equal width 10) (equal length 20)))
> :OKAY

? (retrieve
     (?x
      ((lambda (x y) (* (first x) (first y)))
       (told-value-if-exists (width ?x))
       (told-value-if-exists (length ?x))))
     (?x (and (a width) (a length))))
> (((?x i)
      (((:lambda (x y) (* (first x) (first y)))
        (:existing-told-values (width ?x))
        (:existing-told-values (length ?x)))
       200)))
```

The function / lambda application is performed by substituting the formal parameters `x, y` to the actual arguments supplied by the two `told-value-if-exists` projection operators. These operators return lists of (told) values; thus, the *first* function is applied before `*` is applied to yield the total size.

5.2 Server-Side Programming in nRQL

MiniLisp is easy to understand and use for readers which have some Common Lisp experience. MiniLisp only supports symbols, strings, numbers, and lists. Many of the built-in operators are inherited from CL. All RacerPro API functions can be called from within a lambda body; RacerPro macros are treated as functions.

MiniLisp supports `t'` / `quote`, ` ‘ ` / `backquote`, `,` / `comma`, and `@` / `bq-comma-atsign`. This is very useful if nRQL (sub)queries with variable parts shall be executed from within a lambda body.

The lambda body is evaluated in an environment where `*current-abox*` is bound to the query ABox, and `*current-tbox*` to the query TBox. Moreover, `*output-stream*` is bound to the file output stream within the scope of `with-open-output-file`; this is an ordinary CL output stream, and can thus be used as argument to standard CL functions such as `format`. See below for examples for file output.

The following special forms are provided: `reduce, and, or, not, if, when, unless, cond, maptree, maplist, every, some, progn, prog1, let, let*, lambda, with-nrql-settings`. Note that functions like `maptree` take `lambda` bodies as arguments; however, `lambda` expressions are not first-order in MiniLisp in order to grant termination.

These head projection operators are available as functions: `describe-ind instantiators most-specific-instantiators retrieve-individual-synonyms`.

Some type conversion functions: `to-number to-string to-symbol.`

There are some predefined sorting functions: `sort-string-greaterp sort-string-lessp sort-string< sort-string> sort-symbol-name-greaterp sort-symbol-name-lessp sort< sort>`.

The following functions are borrowed from Common Lisp and work as expected:

`* + - / 1+ 1- < <= = > >= append asin asinh atan atanh ceiling concat cons consp cos cosh eighth ensure-list eq eql equal equalp evenp expt fifth find first flatten float floor format fourth intersection length list listp log max member min minusp ninth nth nth null numberp oddp plusp position princ rationalize remove rest reverse round search second set-difference set-equal set-subset seventh signum sin sinh sixth sqrt string-capitalize string-downcase string-equal string-greaterp string-lessp string-upcase string< string<= string= string> string>= stringp symbol-name symbolp tan tanh tenth terpri third tree-equal type-of union with-open-output-file write zerop`.

5.3 Head-Projection Operators

We have already discussed the told value and attribute retrieval head-projection operators. nRQL offers some more head projection operators which are implemented in MiniLisp. The following head projections operators are implemented in MiniLisp and can be called MiniLisp macro head-projection operators:

- `:all-types`
  (also: `:types` `:instantiators` `:all-instantiators`),
- `:all-types-flat`
  (also: `:all-instantiators-flat`,
  `:types-flat`,
  `:instantiators-flat`),
- `:direct-types`
  (also: `:most-specific-types`,
  `:most-specific-instantiators`,
  `:direct-instantiators`),
- `:direct-types`
  (also: `:most-specific-types`,
  `:most-specific-instantiators`,
  `:direct-instantiators`),
- `:direct-types-flat`
  (also: `:most-specific-types-flat` ,
  `:most-specific-instantiators-flat`,
  `:direct-instantiators-flat`),
- `:describe` as well as
- `:individual-synonyms`.

Here are some examples illustrating these macro operators:

```
? (full-reset)
> :okay-full-reset

? (instance i c)
> :OKAY
```

```
? (implies c d)
> :OKAY

? (retrieve (?x (describe ?x)) (?x top))
> (((?x i)
       (((:lambda (?x) (describe-ind ?x *current-abox*)) ?x)
        (i
         :assertions
         ((i c))
         :role-fillers
         NIL
         :told-attribute-fillers
         NIL
         :told-datatype-fillers
         NIL
         :annotation-datatype-property-fillers
         NIL
         :annotation-property-fillers
         NIL
         :direct-types
         :to-be-computed))))

? (describe-query :last)
> (:query-1
      (:accurate :processed)
      (retrieve
       (?x ((:lambda (?x) (describe-ind ?x *current-abox*)) ?x))
       (?x top)
       :abox
       default))

? (retrieve (?x (types ?x)) (?x top))
> (((?x i)
       (((:lambda (?x) (instantiators ?x *current-abox*)) ?x)
        ((c) (d) (*top* top)))))

? (retrieve (?x (all-types ?x)) (?x top))
> (((?x i)
       (((:lambda (?x) (instantiators ?x *current-abox*)) ?x)
        ((c) (d) (*top* top)))))


? (retrieve (?x (all-types-flat ?x)) (?x top))
> (((?x i)
       (((:lambda
          (?x)
          (sort-symbol-name-lessp
           (flatten (instantiators ?x *current-abox*))))
         ?x)
```

```
        ((c d *top* top)))))
```

Please note that the `retrieve` macro can be advised not to include the lambda expression in the query answer:

```
? (retrieve (?x (all-types-flat ?x)) (?x top) :dont-show-lambdas-p t)
> (((?x i) ((c d *top* top))))
```

It should be noted that the lambda expression can also be used in rule antecedences, as shown below.

In the next sections we present several examples which demonstrate the usefulness of MiniLisp for leveraging the expressivity of query languages.

5.4 Aggregation Operators

First we show how queries with *aggregation operators* can be implemented. Consider the following KB which models the compositional structure of a car. A car has certain parts, and each part has a certain weight:

```
(full-reset)
(define-primitive-role has-part :transitive t)
(define-concrete-domain-attribute weight :type real)

(instance car1 car)
(related car1 engine1 has-part)
(related engine1 cylinder-1-4 has-part)
(related car1 wheel-1-4 has-part)
(related car1 chassis1 has-part)
(instance engine1 (= weight 200.0))
(instance chassis1 (= weight 400.0))
(instance wheel-1-4 (= weight 30.0))
```

Using MiniLisp, we can compute the overall weight as well as identify its number of components:

```
? (retrieve1 (?car car)
           (((lambda (car)
              (let ((w
                     (reduce '+
                             (flatten
                              (retrieve1 `(and (,car car)
                                               (,car ?part has-part)
                                               (?part (a weight)))
                                         `(((lambda (weight) weight)
                                            (told-value-if-exists
                                                (weight ?part))))))))
                    (parts (length (retrieve `(?part) `(,car ?part has-part)))))
                `((?car ,car) (?no-of-parts ,parts) (?total-weight ,w))))
```

```
             ?car)))
> ((((?car car1) (?no-of-parts 4) (?total-weight 630.0))))
```

Please note that `retrieve1` is like `retrieve`, but with head and body argument positions flipped. The body of the query consists of the concept query atom `(?x car)`. The `lambda` expression is then applied to the current binding of `?x`. So, within the `lambda`, `car` is bound to the bindings / value of `?x`. First, the total weight is computed: for this purpose, a subquery is constructed. If `?x = car1`, then the query `(retrieve1 '(and (car1 car) (car1 ?part has-part) (?part (a weight))) ...)` is constructed and posed, asking for the parts of `car1`. The head of the subquery consists of yet another `lambda`, which simply applies the `told-value-if-exists` head projection operator to retrieve the told values of the `weight` attribute of `?part`. The subquery result is returned as a nested list; the list if then flattened, and its items are summed using `(reduce '+ ...)`. The result is bound to the local variable `w`. Similarly, the number of `parts` is computed (by posing yet another subquery). Finally, the result of the `lambda` expression is constructed and returned. The constructed and returned value will become the result tuple. So, if `car` is `car1`, and `no-of-parts` is `4`, and `w` is `630.0`, then the template `'((?car ,car) (?no-of-parts ,parts) (?total-weight ,w))` constructs the result tuple `(((?car car1) (?no-of-parts 4) (?total-weight 630.0)))`.

5.5 Efficient Aggregation Operators using Promises

Although the previous query demonstrated the power and utility of MiniLisp, the aggregation is not computed efficiently. The reason is that for each binding of `?car`, two new subqueries are constructed. Each query has to be parsed, compiled, and is then maintained as a query object. Since the structure of the subqueries does not change during query execution it would be better to construct these subqueries in advance. This can be achieved using so-called *promises*. During the time of query preparation, a promise for a variable declares that this variable is treated as an individual. Moreover, it is promised that at the time when the query is executed, that variable is indeed bound to an individual.

Thus, a more efficient version looks as follows:

```
? (with-future-bindings (?car)
    (delete-all-queries)
    (prepare-abox-query (?part) (and (?car car) (?car ?part has-part))
                  :dont-show-lambdas t
                  :id :parts-of-car)

    (prepare-abox-query (((lambda (weight) weight)
                          (told-value-if-exists (weight ?part))))
            (and (?car car) (?car ?part has-part) (?part (a weight)))
            :dont-show-lambdas t
            :id :weights-of-parts-of-car))
> ...
```

This has defined two queries named `:parts-of-car` and `:weights-of-parts-of-car`. The query optimizer has treated the `?car` variable as an individual due to `with-future-bindings`. Thus, we have promised nRQL that we will only execute these queries if we supply a binding for `agg`. We can establish such a binding during query execution using `with-nrql-settings` as follows:

```
? (retrieve1 (?car car)
   ( ((lambda (car)
        (with-nrql-settings (:dont-show-lambdas t
                             :bindings `((?car ,car)))
           (let ((w
                  (reduce '+
                     (flatten
                       (execute-or-reexecute-query :weights-of-parts-of-car))))
                 (parts (length
                          (execute-or-reexecute-query :parts-of-car))))

             `((?car ,car) (?total-weight ,w)))))
        ?car)))
>  ((((?car car1) (?no-of-parts 4) (?total-weight 630.0))))
```

5.6 User Defined Query Result Format

As already demonstrated, the value returned by a `lambda` body is included in the binding list. In the previous query, the `lambda` body returns a structured list using the template `` `((?car ,car) (?total-weight ,w)) ``; this is equivalent to `(list (list '?car car) (list '?total-weight w))`. It is also easy to specify natural language output; simply replace

```
`((?car ,car) (?total-weight ,w))
```

with

```
(format nil "Car with name  A weights  A kg and has  A parts." car w parts)
```

in the previous query, and you will get:

```
((Car with name car1 weights 630.0d0 kg and has 4 parts.))
```

as the query result.

File Output MiniLisp also offers the `with-open-output-file` which allows to open an output file. Simply print to the stream `*output-stream*` (which is established by `with-open-output-file`) to add arbitrary content to the output file.

```
? (retrieve1 (?car car)
       (((lambda (car)
           (with-open-output-file ("minilisp-output.txt")
              (let ((w
                      (reduce '+
                           (flatten
                            (retrieve1 `(and (,car car)
                                             (,car ?part has-part)
                                             (?part (a weight)))
```

```
                                        `(((lambda (weight) weight)
                                            (told-value-if-exists
                                              (weight ?part))))))))
                        (parts
                          (length
                             (retrieve `(?part) `(,car ?part has-part)))))
                       (format *output-stream*
                               "Car with name ~A weights ~A kg and has ~A parts.~%"
                               car w parts)))))
                    ?car)))
> ((NIL))
```

The output can be found in the file `minilisp-output.txt`; moreover, the result `((NIL))` is delivered because the `lambda` body returns `(NIL)`. Please note that `with-open-output-file` only works if RacerPro in running in unsafe mode.

5.7 Filtering Result Tuples

`lambda` bodies can also work as *filters* as follows: if the special token `:reject` is returned from the `lambda` body, then the result is rejected, i.e., will not appear in the query answer. For example, using `:reject` we can easily reject all parts which have no subparts:

```
? (retrieve1 (?part top)
           (((lambda (part)
              (let ((parts
                      (length
                        (retrieve `(?part) `(,part ?part has-part)))))
                 (if (zerop parts) :reject `(,part has ,parts parts))))
             ?part)))
> (((engine1 has 1 parts)) ((car1 has 4 parts)))
```

5.8 Combined TBox / ABox Queries

Finally, let us present an example with demonstrates how to combine TBox queries and ABox queries. Sometimes, one wants to retrieve all and only the *direct instances* of a concept / OWL class. An individual is called a direct instance of a concept / OWL class if there is no subconcept / subclass of which the individual is also an instance.

Let us create two concepts `c` and `d` such that `d` is a sub concept (child concept) of `c`:

```
? (full-reset)
> :okay-full-reset

? (define-concept c (some r top))
> :OKAY

? (define-concept d (and c e))
> :OKAY
```

We can verify that `d` is indeed a child concept of `c`, using a so-called TBox query, see below:

```
? (tbox-retrieve (?x) (c ?x has-child))
> (((?x d)))
```

Let us create two individuals so that `i` and `j` are instances of `c`; moreover, `j` is also an instance of `d`:

```
? (related i j r)
> :OKAY

? (related j k r)
> :OKAY

? (instance j e)
> :OKAY

? (retrieve (?x) (?x c))
> (((?x j)) ((?x i)))

? (retrieve (?x) (?x d))
> (((?x j)))
```

The previous queries demonstrated that both `i` as well as `j` are `c` instances. However, only `i` is a *direct* `c` instance. We can retrieve these direct instances of `c` as follows:

```
? (retrieve1 (?x c)
      ( ((:lambda (x)
          (if (some (lambda (subclass)
                      (retrieve () `(,x ,subclass)))
                    (flatten
                      (tbox-retrieve1 `(c ?subclass has-child)
                        `( ((:lambda (subclass) subclass) ?subclass)))))
              :reject
           `(?x ,x)))
        ?x)))
> (((?x i)))
```

Here are some MiniLisp examples which demonstrate the points 1. to 3. Consider the following simple ABox:

```
(related i j r)
(related j k r)
```

Suppose you want to create a *comma separated values file* called `test.csv` which contains all the (possible implied) `R` role assertions. MiniLisp allows you to do this:

```
(retrieve (((:lambda (x y)
               (with-open-output-file ("~/test.csv")
                   (format *output-stream* "~A;~A~%" x y)))
            ?x ?y))
      (?x ?y r))
```

So, the query body retrieves all `?x, ?y` individuals which stand in an `R` relationship; for each `?x, ?y` tuple, one more line is attached to the file `test.csv`.

Regarding point 2, let us illustrate how "combined" TBox/ABox queries can be used to retrieve the *direct* instances of a concept, which has been requested by many users.

Let us create two concepts `c` and `d` such that `d` is a sub concept (child concept) of `c`:

```
? (full-reset)
> :okay-full-reset

? (define-concept c (some r top))
> :OKAY

? (define-concept d (and c e))
> :OKAY
```

We can verify that `d` is indeed a child concept of `c`, using a so-called TBox query:

```
? (tbox-retrieve (?x) (c ?x has-child))

> (((?x d)))
```

Let us create two individuals so that `i` and `j` are instances of `c`; moreover, `j` is also an instance of `d`:

```
? (related i j r)
> :OKAY

? (related j k r)
> :OKAY

? (instance j e)
> :OKAY

? (retrieve (?x) (?x c))
> (((?x j)) ((?x i)))

? (retrieve (?x) (?x d))
> (((?x j)))
```

Thus, both `i` and `j` are `c` instances. However, only `i` is a *direct* `c` instance. We can retrieve these direct instances of `c` as follows:

```
? (retrieve1 (?x c)
      ( ((:lambda (x)
          (if (some (lambda (subclass)
                      (retrieve () `(,x ,subclass)))
                    (flatten
                      (tbox-retrieve1 `(c ?subclass has-child)
                        `( ((:lambda (subclass) subclass) ?subclass)))))
              :reject
            `(?x ,x)))
        ?x)))

> (((?x i)))
```

Basically, `retrieve1` is like `retrieve`, but first comes the body, and then the head. Thus, `?x` is bound to a `c` instance. Using this binding, it is checked by means of a TBox subquery (`tbox-retrieve1`) whether the individual bound to `?x` is also an instance of any subclass of `c`. If this is the case, the result tuple (resp. the current binding of `?x`) is *rejected* (see the special `:reject` token); otherwise, the result tuple is *constructed* and returned. The returned result tuples make up the final result set.

Please note that the combination of MiniLisp and `:reject` token gives you the ability to define arbitrary, user-defined *filter predicates* which are executed efficiently since they are directly on the RacerPro server.

Finally, let us consider how *aggregation operators* can be implemented in MiniLisp. Consider the following book store scenario:

```
(full-reset)

(instance b1 book)
(instance b2 book)

(related b1 a1 has-author)
(related b1 a2 has-author)
(related b2 a3 has-author)

(define-concrete-domain-attribute price :type real)
(instance b1 (= price 10.0))
(instance b2 (= price 20.0))
```

We can now determine the number of authors of the single books with the following query:

```
? (retrieve (?x
              ((lambda (book)
                 (let ((authors
                          (retrieve `(?a) `(,book ?a has-author))))
                   (length authors)))
              ?x))
      (?x book)
      :dont-show-lambdas-p t)
```

```
> (((?x b1) 2) ((?x b2) 1))
```

Another interesting question might be to ask for the *average price* of all the books. This is a little bit more tricky, but works in nRQLas well:

```
? (retrieve
      (((lambda nil
         (let ((prices
                (flatten
                  (retrieve `((told-value-if-exists (price ?x)))
                            `(?x book)
                           :dont-show-head-projection-operators-p t))))
           `(average-book-price
               ,(float (/ (reduce '+ prices) (length prices)))))))))
     true-query
    :dont-show-lambdas-p t)

> (((average-book-price 15.0)))
```

The trick here is to use the always true query body `true-query` in order to let nRQLfirst evaluate the lambda body (since lambda bodies are only valid in nRQLheads, but not available as "first order statements" in RacerPro); thus, the lambda simply acquires all the prices of the individual books and then computes and returns the average price, as expected.

Chapter 6

Support for Reasoning with Triples in Secondary Memory

In practical applications, not all parts always require reasoning with expressive Tboxes (specified for instance in OWL ontologies). Indeed, for some purposes, classical data retrieval is just fine. However, in almost all practical applications there will be a large amount of data. Hence, data access must scale at least for the standard retrieval tasks. At the same time, it must be possible to apply expressive reasoning to (parts of) the data without producing copies of the data. For this purpose, we propose to use a triple store for RDF.

6.1 Accessing Triple Stores

Very efficient software is available to query and manipulate RDF triple stores. RacerPro relies on one of the fastest triple store for billions of triples: AllegroGraph from Franz Inc. (`www.franz.com`). The AllegroGraph triple store is part of RacerPro-1.9.1-beta.

Using a triple store has several advantages. On the one hand, triples may be manipulated and retrieved from programs (Turing-complete representations), e.g. Java programs or Common Lisp programs, without reasoning as usual in industrial applications. On the other hand, the same triples can be queried w.r.t. a background ontology. This involves reasoning and might result in additional, implicit triples to be found.

RacerPro allows for accessing existing AllegroGraph triple stores as well as for the creation of new ones. In the following example, an existing triple store is opened, and the triples are read into the knowledge base. Afterwards three nRQL queries are answering over the knowledge base. There is no need to write long-winded data extraction programs that move triples to OWL files on which, in turn, reasoning is then applied.

```
(setf db (open-triple-store "test"))
(use-triple-store db :kb-name 'test-kb)

(retrieve (?x
           (:datatype-fillers (#!name ?x))
           (:datatype-fillers (#!emailAddress ?x))
           (:datatype-fillers (#!telephone ?x)))
  (and (?x #!Professor)
```

```
       (?x |http://www.Department0.University0.edu| #!worksFor)
       (?x (a #!name))
       (?x (a #!emailAddress))
       (?x (a #!telephone))))

(retrieve (?x ?y ?z)
  (and (?x ?y #!advisor)
       (?x ?z #!takesCourse)
       (?y ?z #!teacherOf)
       (?x #!Student)
       (?y #!Faculty)
       (?z #!Course)))

(retrieve (?x ?y)
  (and (?y |http://www.University0.edu| #!subOrganizationOf)
       (?y #!Department)
       (?x ?y #!memberOf)
       (?x #!Chair)))
```

The examples should illustrate the flavor of how triples can be accessed from RacerPro. Currently, for reasoning, the triples are loaded into main memory by RacerPro. Thus, only a limited number of triples should be in the store. In a future version, reasoning will be done also on secondary memory.

As mentioned before, it is not always the case that reasoning is required. Therefore, one can pose the same nRQL queries to secondary memory for very fast access (but without reasoning). RacerPro optimizes the queries in order to provide good average-case performance.

```
(open-triple-store "test")

(pretrieve (?x
           (:datatype-fillers (#!name ?x))
           (:datatype-fillers (#!emailAddress ?x))
           (:datatype-fillers (#!telephone ?x)))
  (and (?x #!Professor)
       (?x |http://www.Department0.University0.edu| #!worksFor)
       (?x (a #!name))
       (?x (a #!emailAddress))
       (?x (a #!telephone))))

(pretrieve (?x ?y ?z)
  (and (?x ?y #!advisor)
       (?x ?z #!takesCourse)
       (?y ?z #!teacherOf)
       (?x #!Student)
       (?y #!Faculty)
       (?z #!Course)))

(pretrieve (?x ?y)
  (and (?y |http://www.University0.edu| #!subOrganizationOf)
```

```
        (?y #!Department)
        (?x ?y #!memberOf)
        (?x #!Chair)))
```

In addition, it is possible to materialize in a triple store what can be computed by applying reasoning w.r.t. a background ontology such that later on the results are available to all applications which may or may not use reasoning. It is possible to optimized index data structures for subsequent query answering.

```
(setf db (open-triple-store "test"))
(use-triple-store db :kb-name 'test-kb :ignore-import t)
(materialize-inferences 'test-kb :db db :abox t :index-p t)
```

A triple store may be created by RacerPro as well.

```
(setf db (create-triple-store "test" :if-exists :supersede))
(triple-store-read-file "..." :db db)
```

6.2 Supporting Standards: SPARQL

Queries can also be specified in the SPARQL language and can be executed with our without reasoning w.r.t. background ontologies.

```
(sparql-retrieve "
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x, ?y
WHERE
 ( ?x lubm:subOrganizationOf "http://www.University0.edu" )
 ( ?y rdf:type lubm:Department )
 ( ?x lubm:memberOf ?y )
 ( ?x rdf:type lubm:Chair )
")
```

This is the SPARQL pendant to the third query in the example above.

Note that there is no need to store the OWL ontology in the triple store. With RacerPro you can access your existing triple store with various different OWL ontologies. An ontology can be put into a file or can possibly be retrieved from the Web. A triple store is then opened and queries are answered w.r.t. this triple store. The triple store corresponds to the Abox in this case.

Chapter 7

Representing Probabilistic Knowledge

7.1 Preliminaries: Linear Programming

Here the foundations of the theory, presented in Chapter 7.2, are introduced. It is **not** intended to do complete and comprehensive discussions of these topics.

Linear Programming in a "mathematical nutshell" is the solution of finite systems of linear equations and inequalities in a finite number of variables [32].

A linear program is defined as follows:

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{7.1a}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_j^i x_j \geq b_i \quad \text{for } i = 1, \ldots, m \tag{7.1b}$$

$$x_j \geq 0 \quad \text{for } j = 1, \ldots, m \tag{7.1c}$$

Formula 7.1a is called the *objective function* of the linear program. The goal is to compute its maximum (respectively minimum) with respect to the set of linear constraints given in 7.1b. Usually the considered variables x_j have to be non-negative (7.1c).

There are three possible solutions to a linear program:

1. the solution is unfeasible, meaning the linear program has no solution at all
2. the solution is unbounded, meaning the solution goes to infinity
3. the solution is the optimal one.

Linear programs may be applied to a wide range of problems in planning and scheduling, e.g. in business and economics.

In the late 1940's, George B. Dantzig [5], a mathematician at the US Air Force, invented the so called *simplex method* [6]. This invention started off research in linear programming and its applications, which ignited the field of operations research. In 1972 the worst case behaviour of the algorithm was shown by Klee and Minty to have exponential complexity. Ten years later

Borgwardt was able to proof that its average case behaviour is of polynomial complexity [2]. In recent years, the simplex method performance has been still a matter of research interest [39].

A new algorithm was proposed in 1979, which owns polynomial worst case behaviour [32]. It is called the *ellipsoid algorithm*. However, this algorithm has convergence problems due to numerical computation.

The *interior point algorithm* was devised in 1984, providing both polynominal complexity and numerical stability. Nevertheless improved simplex algorithms remain used widely in practical applications.

Software packages implementing algorithms to solve linear programs are called *linear program solvers*. There are numerous textbooks available on linear programming [32, 2, 11].

7.2 Extensions to Description Logics

This section gives an overview over the research related and presents two theories required for the formalization of probabilistic reasoning in description logics. The research related to the theories under investigation may be divided into two categories: *probabilistic logics* on one hand and *non-monotonic logics* on the other. The first category deals with the combination of probability theory and logic, the second with exceptions in the logic.

7.2.1 Logics and Probabilities

In "Boole's Logic and Probability", Halperin [15] investigates the research George Boole(1815-1864) did on the combination of probability theory and logic. He also describes, how Boole's work may be applied to the subject of linear programming. Thus we may state that combining logics and probability is rather an old field of research.

Several probabilistic reasoning methods for knowledge based systems have been discussed in books by Pearl[33], Bacchus [1], and Russel and Norvig [37]. In the literature two approaches can be found in order to perform inferences with probabilities. The first approach uses Bayesian or believe networks to determine the inferred probabilities [33, 37], the second makes use of linear programming for this task.

Both approaches previously mentioned have been tried to be combined with description logics. The usage of Bayesian networks has been applied in [19, 8] in order to achieve the probabilistic extension. In [17] the inference could possibly be made by linear programming. The paper under investigation [12] and its follow up [22] are using linear programming for probabilistic inference.

7.2.2 Non-monotonic Logics

Both theories, which will be introduced in the next sections, may be categorised as non-monotonic logics. The difference between monotonic and non-monotonic reasoning is summarised below.

Reasoning in Predicate or First Order Logic is monotone with respect to the available knowledge. More precisely adding new statements to the knowledge base "can only increase the set of entailed sentences" [37] or in other words "if something is known to be *true* (*false*), it will never become *false* (*true*)" [14]. In [37]]this is formally stated as:

$$\text{If} \qquad KB \models \phi \qquad \text{then} \qquad KB \wedge \psi \models \phi \tag{7.2}$$

By the observation of human reasoning one recognises that it is not monotone at all. This can be informally described as "changing one's mind". Humans allow exceptions in their knowledge which can not be resembled neither in Predicate nor in First Order Logic. To address these shortcomings several non-monotonic reasoning methods [36, 24, 25] have been developed.

Default Logic is a non-monotonic reasoning method developed by Reiter [36]. It is also discussed in [33, 37]. In this approach one differentiates between hard facts about a world and default rules. "Since the set of all hard facts cannot completely specify the world, we are left with gaps in our knowledge; the purpose of the default rules is to help fill in those gaps with plausible conclusions" [33].

Further research in default logic developed several different kinds of entailments, e.g. 0-entailment, 1-entailment, z-entailment, lexicographic entailment and me-entailment to name a few (see [3] for an overview).

Lexicographic entailment[20] and its probabilistic successors[23, 21, 13, 22] will be the object of theory investigation in this section.

7.3 Lexicographic Entailment

The notion of *lexicographic entailment* was first proposed by Lehmann [20] for sets of defaults. As an extension of Pearls System Z [34] it allows for inheritance to exceptional "subclasses".

To get a better understanding of the discussed topic let us discuss an example (called "Winged Penguins" in literature [20, 3]). It is also used to illustrate the terms needed. The syntax is the one used by Lehmann.

Consider the following set of defaults D:

$$\mathrm{D} = \{(b : f), (b : w), (p : b), (p : \neg f)\} \tag{7.3}$$

This set of defaults D may be interpreted as follows:

BIRDS(b)FLY(f), BIRDS HAVE WINGS(w), PENGUINS(p) ARE BIRDS BUT PENGUINS DO NOT FLY.

Defaults represent knowledge about how things generally are, i.e *birds fly*. Lehmann chooses the presumptive reading of a default to give its meaning: "Birds are presumed to fly unless there is evidence to the contrary." He denotes defaults as $(\phi : \psi)$ where ϕ and ψ are formulae in propositional logic. The meaning of a default $(\phi : \psi)$ is closely related to the implication $\phi \Rightarrow \psi$, also called the defaults material counterpart. The interpretation is given as statements of high conditional probability $P(\psi|\phi) \geq 1 - \epsilon$ [34].

If we would have used a propositional knowledge base in our example instead of defaults it would be inconsistent and therefore entail anything, e.g. penguins fly and penguins don't fly. The goal of the definition of the lexicographic entailment is to regain a consistent knowledge base, a knowledge base where we only are able to conclude that penguins don't fly.

We achieve this goal by applying a lexicographic ordering onto our models in order to consider only preferred or more natural models. We give reasons for doing so shortly, but for now let's return to the "Winged Penguins" example.

In propositional logic a truth assignment to a primitive proposition is called a *model* (or *world*). As our example shows we have four primitive propositions b, f, p and w. Because of that there are 2^4 possible models denoted m_i with $i \in \{1, \ldots, 16\}$. See table 7.1[1] for further insight.

[1] In this table 0 represents $false$ and 1 represents $true$; except for the columns D_0 and D_1, they represent the

m	b	f	p	w	$b \Rightarrow f$	$b \Rightarrow w$	$p \Rightarrow b$	$p \Rightarrow \neg f$	D_0	D_1	$p \wedge w$	$p \wedge \neg w$
m_1	0	0	0	0	1	1	1	1	2	2	0	0
m_2	0	0	0	1	1	1	1	1	2	2	0	0
m_3	0	0	1	0	1	1	0	1	2	1	0	1
m_4	0	0	1	1	1	1	0	1	2	1	1	0
m_5	0	1	0	0	1	1	1	1	2	2	0	0
m_6	0	1	0	1	1	1	1	1	2	2	0	0
m_7	0	1	1	0	1	1	0	0	2	0	0	1
m_8	0	1	1	1	1	1	0	0	2	0	1	0
m_9	1	0	0	0	0	0	1	1	0	2	0	0
m_{10}	1	0	0	1	0	1	1	1	1	2	0	0
m_{11}	1	0	1	0	0	0	1	1	0	2	0	1
m_{12}	1	0	1	1	0	1	1	1	1	2	1	0
m_{13}	1	1	0	0	1	0	1	1	1	2	0	0
m_{14}	1	1	0	1	1	1	1	1	2	2	0	0
m_{15}	1	1	1	0	1	0	1	0	1	1	0	1
m_{16}	1	1	1	1	1	1	1	0	2	1	1	0

Table 7.1: Winged Penguins

A model m *satisfies* a propositional formula ϕ if and only if $m(\phi) = true$. This is represented by $m \models \phi$. A default $(\phi : \psi)$ is satisfied by a model, denoted $m \models (\phi : \psi)$, if and only if $m \models \phi \Rightarrow \psi$.

Further a model is said to *verify* (or as the opposite *falsify*) a default $(\phi : \psi)$ if and only if $m \models \phi \wedge \psi$ (or $m \models \phi \wedge \neg\psi$). One default is *tolerated* by a set of defaults D iff there exists a model which verifies it and simultaneously satisfies the others. If no such model exists, the default is said to be *in conflict* with the other defaults.

In the example one can see that the default $(b : f)$ is tolerated by D. For that purpose we first have a look at the models $m_{13} - m_{16}$ which verify the default. Afterwards m_{14} can be identified to satisfy all remaining defaults. m_{14} also shows that the default $(b : w)$ is tolerated by D. If we inspect the models $m_{11}, m_{12}, m_{15}, m_{16}$, which are the ones verifying the default $(p : b)$, then none of them satisfies all other three defaults at the same time. Therefore $(p : b)$ is in conflict with D. Similarly the reader may show the conflict of $(p : \neg f)$.

The *z-partition* of the set of defaults D is defined as the unique ordered partition $(\mathrm{D}_0, \ldots, \mathrm{D}_k)$ such that each D_i with $i \in \{0, \ldots, k\}$ contains the set of all defaults that are tolerated under $\mathrm{D} \setminus (\mathrm{D}_0 \cup \cdots \cup \mathrm{D}_{i-1})$.

Above we did the first step for computing the z-partition of our example. Only the defaults $(b : f)$ and $(b : w)$ were tolerated by all other defaults. Thus they build D_0. D_0 is then removed from D. The remaining two defaults tolerate each other. Hence they are contained in D_1. We now have the following z-partition $(\mathrm{D}_0, \mathrm{D}_1)$ where:

$$\mathrm{D}_0 = \{(b : f), (b : w)\} \qquad \text{and} \qquad \mathrm{D}_1 = \{(p : b), (p : \neg f)\} \tag{7.4}$$

If the unique z-partition can be build the set of defaults D is said to be *consistent*.

The z-partition groups the defaults by specificity. A part of the z-partition is more specific than another if its index is higher. For example the defaults $(p : b)$, $(p : \neg f)$ in part D_1 are the most specific ones.

count of satisfied defaults in the subset D_i of the z-partition.

With possible conflicting defaults and the inability to find a model that tolerates all of them, the goal is to select a preferred model to draw the conclusion from. Because of this two criteria have to be taken into account:

1. A model is preferred to another model if it satisfies more defaults.
2. Satisfying a more specific default is preferred to a less specific one.

Both criteria allow us to order the models according to them. Both orderings should be combined into one, in order to determine the preferred model. A lexicographic ordering provides such an integration of orderings. The second criterion is chosen as the major criterion, since it is preferred to satisfy a more specific default than to satisfy less specific ones.

Using the z-partition as a degree of specificity the lexicographic order for the models is defined as follows:

A model m is called *lexicographical preferred* to a model m' if and only if some $i \in \{0, \ldots, k\}$ can be found that $|\{d \in \mathrm{D}_i \mid m \models d\}| > |\{d \in \mathrm{D}_i \mid m' \models d\}|$ and $|\{d \in \mathrm{D}_j \mid m \models d\}| = |\{d \in \mathrm{D}_j \mid m' \models d\}|$ for all $i < j \leq k$.

Or in other words we are counting the defaults satisfied in each part of the z-partition for every model. Then starting with the k th subset of the z-partition and comparing which model satisfies more defaults d in that subset. If there can not be made a decision we have to look at the $k-1$ subset and make the comparison and so on.

In the example we have the following lexicographic order on the models: $m_1, m_2, m_5, m_6, m_{14} > m_{10}, m_{12}, m_{13} > m_9, m_{11} > m_3, m_4, m_{16} > m_{15} > m_7, m_8$.

m is said to be the *lexicographically minimal* model if and only if no other models are lexicographically preferable to m. Lexicographic entailment is defined by comparing the minimal verifying and falsifying models of a default $(\phi : \psi)$.

In order to determine if the default $(p : w)$, which can be interpreted as PENGUINS HAVE WINGS, is lexicographically entailed in D, one has to look at the models that verify or falsify the default $(p : w)$ and therefore have to satisfy $p \wedge w$ and do not satisfy $p \wedge \neg w$ or vice versa, here the models $m_3, m_4, m_7, m_8, m_{11}, m_{12}, m_{15}, m_{16}$. The lexicographically minimal model is m_{12}. This model verifies the default $(p : w)$ and therefore PENGUINS HAVE WINGS is lexicographically entailed in D.

7.4 Probabilistic Lexicographic Entailment for Description Logics

In order to extend a Description Logic for dealing with probabilistic knowledge an additional syntactical and semantical construct is needed. This additional construct is called a *conditional constraint*. It is an extension of defaults, introduced in the previous section, which represent knowledge that is generally true.

This extension of description logics has been first formalized by Giugno and Lukasiewicz in 2002 [12, 13]. All following definitions are given in accordance to their paper.

A conditional constraint consists of a statement of conditional probability for two concepts C, D as well as a lower bound l and an upper bound u constraint on that probability. It is written as follows:

$$(D|C)[l, u] \tag{7.5}$$

Where C can be called the *evidence* and D the *hypothesis*

To gain the ability to store such statements in a knowledge base it has to be extended to a probabilistic knowledge base PKB here named probabilistic terminology $\mathcal{P}$. Additionally to the TBox T here classical terminology $\mathcal{T}_g$ of a description logic knowledge base we introduce the PTBox PT here generic probabilistic terminology $\mathcal{P}_g$, which consists of $\mathcal{T}_g$ and a set of conditional constraints $\mathcal{D}_g$, and a PABox P_o or assertional probabilistic terminology $\mathcal{P}_o$ holding conditional constraints for every probabilistic individual o. In [13] there is no ABox declared, knowledge about so called classical individuals is also stored inside the TBox abusing nominals.

$\mathcal{D}_g$ therefore represents statistical knowledge about concepts and P_o represents degrees of belief about the individuals o.

To be able to define the semantics for a Description Logic with probabilistic extension the interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot)$ has to be extended by a probability function μ on the domain of interpretation $\Delta^{\mathcal{I}}$. The extended interpretation is called the *probabilistic interpretation* $\mathcal{P}r = (\mathcal{I}, \mu)$. Each individual o in the domain $\Delta^{\mathcal{I}}$ is mapped by the probability function μ to a value in the interval [0,1] and the values of all $\mu(o)$ have to sum up to 1 for any probabilistic interpretation $\mathcal{P}r$.

With the probabilistic interpretation $\mathcal{P}r$ at hand the probability of a concept C, represented by $\mathcal{P}r(C)$, is defined as sum of all $\mu(o)$ where $o \in C^{\mathcal{I}}$.

The probabilistic interpretation of a conditional probability $\mathcal{P}r(D|C)$ is given as

$$\frac{\mathcal{P}r(C \sqcap D)}{\mathcal{P}r(C)} \tag{7.6}$$

where $\mathcal{P}r(C) > 0$.

In order to achieve a probabilistic lexicographic entailment all the terms known from the previous section 7.3 have to be redefined for the context of Description Logics.

A conditional constraint $(D|C)[l, u]$ is *satisfied* by $\mathcal{P}r$ or $\mathcal{P}r$ *models* $(D|C)[l, u]$ if and only if $\mathcal{P}r(D|C) \in [l, u]$. We will write this as $\mathcal{P}r \models (D|C)[l, u]$. A probabilistic interpretation $\mathcal{P}r$ is said to *satisfy* or *model* a terminology axiom T, written $\mathcal{P}r \models T$, if and only if $\mathcal{I} \models T$. A set $\mathcal{F}$ consisting of terminological axioms and conditional constraints, where F denotes the elements of $\mathcal{F}$, is satisfied or modeled by $\mathcal{P}r$ if and only if $\mathcal{P}r \models F$ for all $F \in \mathcal{F}$.

The *verification* of a conditional constraint $(D|C)[l, u]$ is defined as $\mathcal{P}r(C) = 1$ and $\mathcal{P}r$ has to be a model of $(D|C)[l, u]$. We also may say $\mathcal{P}r$ *verifies* the conditional constraint $(D|C)[l, u]$. On the contrary the *falsification* of a conditional constraint $(D|C)[l, u]$ is given if and only if $\mathcal{P}r(C) = 1$ and $\mathcal{P}r$ does **not** satisfy $(D|C)[l, u]$. It is also said that $\mathcal{P}r$ *falsifies* $(D|C)[l, u]$.

Further a conditional constraint F is said to be *tolerated* under a Terminology $\mathcal{T}$ and a set of conditional constraints $\mathcal{D}$ if and only if a probabilistic interpretation $\mathcal{P}r$ can be found that verifies F and $\mathcal{P}r \models \mathcal{T} \cup \mathcal{D}$.

With all these definitions at hand we are now prepared to define the *z-partition* of a set of generic conditional constraints $\mathcal{D}_g$. The z-partition is build as ordered partition $(\mathcal{D}_0, \ldots, \mathcal{D}_k)$ of $\mathcal{D}_g$, where each part $\mathcal{D}_i$ with $i \in \{0, \ldots, k\}$ is the set of all conditional constraints $F \in \mathcal{D}_g \backslash (\mathcal{D}_0 \cup \cdots \cup \mathcal{D}_{i-1})$, that are tolerated under the generic terminology $\mathcal{T}_g$ and $\mathcal{D}_g \backslash (\mathcal{D}_0 \cup \cdots \cup \mathcal{D}_{i-1})$.

If the z-partition can be build from a generic probabilistic terminology $\mathcal{P}_g = (\mathcal{T}_g, \mathcal{D}_g)$, it is said to be *generically consistent* or *g-consistent*. A probabilistic terminology $\mathcal{P} = (\mathcal{P}_g, (\mathcal{P}_o)_{o \in I_p})$ is *consistent* if and only if $\mathcal{P}_g$ is g-consistent and $\mathcal{P}r \models \mathcal{T}_g \cup \mathcal{T}_o \cup \mathcal{D}_o \cup \{(\{o\}|\top)[1, 1]\}$ for all $o \in I_p$.

Once more we use the z-partition for the definition of the lexicographic order on the probabilistic

interpretations $\mathcal{P}r$ as follows:

A probabilistic interpretation $\mathcal{P}r$ is called *lexicographical preferred* to a probabilistic interpretation $\mathcal{P}r'$ if and only if some $i \in \{0, \ldots, k\}$ can be found, that $|\{F \in \mathrm{D}_i \mid \mathcal{P}r \models F\}| > |\{F \in \mathrm{D}_i \mid \mathcal{P}r' \models F\}|$ and $|\{F \in \mathrm{D}_j \mid \mathcal{P}r \models F\}| = |\{F \in \mathrm{D}_j \mid \mathcal{P}r' \models F\}|$ for all $i < j \leq k$.

We say a probabilistic interpretation $\mathcal{P}r$ of a set $\mathcal{F}$ of terminological axioms and conditional constraints is a *lexicographically minimal model* of $\mathcal{F}$ if and only if no probabilistic interpretation $\mathcal{P}r'$ is lexicographical preferred to $\mathcal{P}r$.

By now the meaning of *lexicographic entailment* for conditional constraints from a set $\mathcal{F}$ of terminological axioms and conditional constraints under a generic probabilistic terminology $\mathcal{P}_g$ is given as:

A conditional constraint $(D|C)[l, u]$ is a *lexicographic consequence* of a set $\mathcal{F}$ of terminological axioms and conditional constraints under a generic probabilistic terminology $\mathcal{P}_g$, written as $\mathcal{F} \mid\!\sim (D|C)[l, u]$ under $\mathcal{P}_g$, if and only if $\mathcal{P}r(D) \in [l, u]$ for every lexicographically minimal model $\mathcal{P}r$ of $\mathcal{F} \cup \{(C|\top)[1, 1]\}$. *Tight lexicographic consequence* of $\mathcal{F}$ under $\mathcal{P}_g$ is defined as $\mathcal{F} \mid\!\sim_{tight} (D|C)[l, u]$ if and only if l is the infimum and u is the supremum of $\mathcal{P}r(D)$. We define $l = 1$ and $u = 0$ if **no** such probabilistic interpretation $\mathcal{P}r$ exists.

Finally we define lexicographic entailment using a probabilistic terminology $\mathcal{P}$ for generic and assertional conditional constraints F.

If F is a generic conditional constraint, then it is said to be a lexicographic consequence of $\mathcal{P}$, written $\mathcal{P} \mid\!\sim F$ if and only if $\emptyset \mid\!\sim F$ under $\mathcal{P}_g$ and a tight lexicographic consequence of $\mathcal{P}$, written $\mathcal{P} \mid\!\sim_{tight} F$ if and only if $\emptyset \mid\!\sim_{tight} F$ under $\mathcal{P}_g$.

If F is an assertional conditional constraint for $o \in I_P$, then it is said to be a lexicographic consequence of $\mathcal{P}$, written $\mathcal{P} \mid\!\sim F$, if and only if $\mathcal{T}_o \cup \mathcal{D}_o \mid\!\sim F$ under $\mathcal{P}_g$ and a tight lexicographic consequence of $\mathcal{P}$, written $\mathcal{P} \mid\!\sim_{tight} F$ if and only if $\mathcal{T}_o \cup \mathcal{D}_o \mid\!\sim_{tight} F$ under $\mathcal{P}_g$.

7.4.1 Computing Tight Lexicographic Consequence

Now we will introduce the techniques required to compute g-consistency and tight lexicographic consequence by means of an example. The general approach in solving these two problems is to use standard reasoning provided by a description logic reasoner in order to build linear programs which may be handed over to a solver in order to decide the solvability of the programs. Let us consider the following Terminology $\mathcal{T}_g$ and set of conditional constraints $\mathcal{D}_g$:

$$\mathcal{T}_g = \{P \sqsubseteq B\} \qquad \mathcal{D}_g = \{(W|B)[.95, 1]; (F|B)[.9, .95]; (F|P)[0, .05]\} \tag{7.7}$$

Again we may interpret the concepts as follows: P stands for penguins, B for birds, F for flying objects and W for winged objects.

Deciding Satisfiability

The first objective is to build a set $\mathcal{R}_{\mathcal{T}}(\mathcal{F})$. It contains the mappings r, which map conditional constraints $F_i = (D_i|C_i)[l_i, u_i]$ elements of a set of conditional constraints $\mathcal{F}$ onto one of the following terms $D_i \sqcap C_i$, $\neg D_i \sqcap C_i$ or $\neg C_i$ under the condition that the intersection of our mappings r is not equal to the bottom concept given the terminology $\mathcal{T}$, written $\mathcal{T} \not\models r(F_1) \sqcap \cdots \sqcap r(F_n) \sqsubseteq \bot$.

All possible mappings of our example are displayed in Table 7.2, columns 2 to 4. In the first column the intersection of the mappings r is build. As an abbreviation we will write $\sqcap r$ instead of $r(F_1) \sqcap \cdots \sqcap r(F_n)$. The rows of the first column contain a bottom concept $\bot$ in case the intersections of r are empty. Also the concepts causing the bottom concept are given. In the fifth column the $\sqcap r$ are tested against the terminology. If the intersections of r are **not** the same as the bottom concept $\bot$, this is denoted by $\models$, else it is denoted by $\bot$.

$\sqcap r$	$r((W\|B)[.95, 1])$	$r((F\|B)[.9, .95])$	$r((F\|P)[0, .05])$	$\mathcal{T} \not\models \sqcap r \sqsubseteq \bot$
$B \sqcap P \sqcap W \sqcap F$	$W \sqcap B$	$F \sqcap B$	$F \sqcap P$	$\models$
$B \sqcap P \sqcap \neg W \sqcap F$	$\neg W \sqcap B$	$F \sqcap B$	$F \sqcap P$	$\models$
$\bot, \neg B \sqcap B$	$\neg B$	$F \sqcap B$	$F \sqcap P$	
$\bot, \neg F \sqcap F$	$W \sqcap B$	$\neg F \sqcap B$	$F \sqcap P$	
$\bot, \neg F \sqcap F$	$\neg W \sqcap B$	$\neg F \sqcap B$	$F \sqcap P$	
$\bot, \neg B \sqcap B$	$\neg B$	$\neg F \sqcap B$	$F \sqcap P$	
$\bot, \neg B \sqcap B$	$W \sqcap B$	$\neg B$	$F \sqcap P$	
$\bot, \neg B \sqcap B$	$\neg W \sqcap B$	$\neg B$	$F \sqcap P$	
$\neg B \sqcap P \sqcap F$	$\neg B$	$\neg B$	$F \sqcap P$	$\bot$
$\bot, \neg F \sqcap F$	$W \sqcap B$	$F \sqcap B$	$\neg F \sqcap P$	
$\bot, \neg F \sqcap F$	$\neg W \sqcap B$	$F \sqcap B$	$\neg F \sqcap P$	
$\bot, \neg B \sqcap B$	$\neg B$	$F \sqcap B$	$\neg F \sqcap P$	
$B \sqcap P \sqcap W \sqcap \neg F$	$W \sqcap B$	$\neg F \sqcap B$	$\neg F \sqcap P$	$\models$
$B \sqcap P \sqcap \neg W \sqcap \neg F$	$\neg W \sqcap B$	$\neg F \sqcap B$	$\neg F \sqcap P$	$\models$
$\bot, \neg B \sqcap B$	$\neg B$	$\neg F \sqcap B$	$\neg F \sqcap P$	
$\bot, \neg B \sqcap B$	$W \sqcap B$	$\neg B$	$\neg F \sqcap P$	
$\bot, \neg B \sqcap B$	$\neg W \sqcap B$	$\neg B$	$\neg F \sqcap P$	
$\neg B \sqcap P \sqcap \neg F$	$\neg B$	$\neg B$	$\neg F \sqcap P$	
$B \sqcap \neg P \sqcap W \sqcap F$	$W \sqcap B$	$F \sqcap B$	$\neg P$	$\models$
$B \sqcap \neg P \sqcap \neg W \sqcap F$	$\neg W \sqcap B$	$F \sqcap B$	$\neg P$	$\models$
$\bot, \neg B \sqcap B$	$\neg B$	$F \sqcap B$	$\neg P$	
$B \sqcap \neg P \sqcap W \sqcap \neg F$	$W \sqcap B$	$\neg F \sqcap B$	$\neg P$	$\models$
$B \sqcap \neg P \sqcap \neg W \sqcap \neg F$	$\neg W \sqcap B$	$\neg F \sqcap B$	$\neg P$	$\models$
$\bot, \neg B \sqcap B$	$\neg B$	$\neg F \sqcap B$	$\neg P$	
$\bot, \neg B \sqcap B$	$W \sqcap B$	$\neg B$	$\neg P$	
$\bot, \neg B \sqcap B$	$\neg W \sqcap B$	$\neg B$	$\neg P$	
$\neg B \sqcap \neg P$	$\neg B$	$\neg B$	$\neg P$	$\models$

Table 7.2: $\mathcal{DL}$-Winged Penguins

Our set $\mathcal{R}_\mathcal{T}(\mathcal{F})$ contains the following nine mappings r,

$\mathcal{R}_\mathcal{T}(\mathcal{F}) = \{$
$\{W \sqcap B, F \sqcap B, F \sqcap P\}, \{\neg W \sqcap B, F \sqcap B, F \sqcap P\},$
$\{W \sqcap B, \neg F \sqcap B, \neg F \sqcap P\}, \{\neg W \sqcap B, \neg F \sqcap B, \neg F \sqcap P\},$
$\{W \sqcap B, F \sqcap B, \neg P\}, \{\neg W \sqcap B, F \sqcap B, \neg P\},$
$\{W \sqcap B, \neg F \sqcap B, \neg P\}, \{\neg W \sqcap B, \neg F \sqcap B, \neg P\},$
$\{\neg B, \neg B, \neg P\}$
$\}$

With the set $\mathcal{R}_\mathcal{T}(\mathcal{F})$ at hand we are able to set up linear programs to decide the satisfiability of the terminology $\mathcal{T}$ and a finite set of conditional constraints $\mathcal{F}$. We say that $\mathcal{T} \cup \mathcal{F}$ is satisfiable if and only if the linear program 7.8 is solvable for variables y_r, where $r \in \mathcal{R}_\mathcal{T}(\mathcal{F})$. This means, that in the objective function all coefficients preceding the variables y_r are set to 1.

$$\sum_{r\in R, r\models\neg D\sqcap C} -ly_r + \sum_{r\in R, r\models D\sqcap C} (1-l)y_r \geq 0 \quad \text{(for all } (D|C)[l,u]\in\mathcal{F}) \tag{7.8a}$$

$$\sum_{r\in R, r\models\neg D\sqcap C} uy_r + \sum_{r\in R, r\models D\sqcap C} (u-1)y_r \geq 0 \quad \text{(for all } (D|C)[l,u]\in\mathcal{F}) \tag{7.8b}$$

$$\sum_{r\in R} y_r = 1 \tag{7.8c}$$

$$y_r \geq 0 \quad \text{(for all } r\in R) \tag{7.8d}$$

To be able to set up linear programs we further need to introduce the meaning of $r \models C$ which is used as index of the summation in 7.8a and 7.8b. It is an abbreviation for $\emptyset \models \sqcap r \sqsubseteq C$. So the coefficient preceding the variables y_r is set in linear constraints 7.8a and 7.8b if either $r \models \neg D \sqcap C$ or $r \models D \sqcap C$ may be proven.

Computing g-Conscistency

With the tool at hand to decide satisfiability, we may also decide, if a conditional constraint may be tolerated by a set of conditional constraints $\mathcal{F}$. To verify a constraint we add a conditional constraint $(C|\top)[1,1]$. With the extended set the linear program is generated and solved. If an unfeasible solution is computed the conditional constraint is conflicting. If an optimal solution is found, the conditional constraint is tolerated.

Now the z-partition of a set of conditional constraints is computable. For the example we would have to set up four linear programs to compute part D_0 and two for D_1.

Computing Tight Logical Entailment

How to compute tightest probability bounds for given evidence C and conclusion D in respect to a set of conditional constraints $\mathcal{F}$ under a terminology $\mathcal{T}$ is explained in the following text. The task is named *tight logical entailment* and denoted $\mathcal{T}\cup\mathcal{F} \models_{tight} (D|C)[l,u]$.

Given that $\mathcal{T}\cup\mathcal{F}$ is satisfiable, a linear program is set up for $\mathcal{F}\cup(C|\top)[1,1]\cup(D|\top)[0,1]$. The objective function is set to $\sum_{r\in R, r\models D} y_r$. So the coefficient in front of the variables y_r are set 1 if $r \models D$.

The tight logical entailed lower bound l is computed by minimising respectively the upper bound u by maximising the linear program.

Computing Probabilistic Lexicographic Entailment

In order to compute tight probabilistic lexicographic entailment for given evidence C and conclusion D under a generic probabilistic terminology $\mathcal{P}_\}$ and a set $\mathcal{F}$ of terminological axioms and conditional constraints the following steps have to be taken:

1. Compute the z-partition of $\mathcal{D}_g$ in order to be able to generate a lexicographic ordering
2. Compute lexicographic minimal sets $\mathcal{D}'$ of conditional constraints of $\mathcal{D}_g$ as elements of $\overline{\mathcal{D}}$.
3. Compute the tight logical entailment $\mathcal{T}\cup\mathcal{F}\cup\mathcal{D}' \models_{tight} (D|C)[l,u]$ for all $\mathcal{D}'\in\overline{\mathcal{D}}$.

4. Select the minimum of all computed lower bounds and the maximum of all upper bounds.

The 2. step needs some explanation since a new task "compute *lexicographic minimal sets*" is introduced. In order to define a lexicographic minimal set $\mathcal{D}'$, a preparatory definition is required.

A set $\mathcal{D}' \subset \mathcal{D}_g$ *lexicographic preferable* to $\mathcal{D}'' \subset \mathcal{D}_g$ if and only if some $i \in \{0, \ldots, k\}$ exists such that $|\mathcal{D}' \cap \mathcal{D}_i| > |\mathcal{D}'' \cap \mathcal{D}_i|$ and $|\mathcal{D}' \cap \mathcal{D}_i| > |\mathcal{D}'' \cap \mathcal{D}_i|$ for all $i < j \leq k$.

With the lexicographic order introduced onto the sets $\mathcal{D}'$ the definition of lexicographic minimal is given as:

$\mathcal{D}'$ is lexicographic minimal in $\mathcal{S} \subseteq \{S | S \subseteq \mathcal{D}_g\}$ if and only if $\mathcal{D}' \in \mathcal{S}$ and no $\mathcal{D}'' \in \mathcal{S}$ is lexicographic preferable to $\mathcal{D}'$.

Note that equivalence of using lexicographic minimal sets instead of lexicographically minimal models is not proven in [12]. A prove is given in the follow up [22].

7.4.2 Using Probabilistic Lexicographic Entailment

The probabilistic lexicographic entailment may be used to decide the following two interesting probabilistic inference problems:

Probabilistic Subsumption

To compute a probabilistic subsumption for the concepts D, C the tight probabilistic lexicographic entailment is determined with respect to $\mathcal{P}_g$ and $\mathcal{F} = \emptyset$.

Probabilistic Instance Checking

Probabilistic instance checking for an individual o and a concept E is done by computing the tight probabilistic lexicographic entailment with E set as conclusion and $\top$ as evidence and with respect to $\mathcal{P}_g$ and $\mathcal{F} = \mathcal{P}_o$.

7.5 Application Example

Let us consider an annotated image (Fig. 2.1) and a modified athletics ontology from the BOEMIE project to demonstrate how the new services can be applied in this context. Concept definitions have been modeled as conditional constraints in the PTBox while the concept hierarchy remains in the TBox. Relevant axioms and conditional constraints for our example are shown below:

$\mathcal{T} = \{$	$\mathcal{P}_g = \{$
$High\_Jump \sqsubseteq Jumping\_Event$	$(High\_Jump \mid \exists hasPart.Bar)[0.4, 1.0]$
$Pole\_Vault \sqsubseteq Jumping\_Event$	$(Pole\_Vault \mid \exists hasPart.Bar)[0.4, 1.0]$
$High\_Jump \sqcap Pole\_Vault \sqsubseteq \bot\}$	$(Pole\_Vault \mid \exists hasPart.Pole)[0.7, 1.0]$
	$(Pole\_Vault \mid \exists hasPart.Bar \sqcap \exists hasPart.Pole)[0.8, 1.0]\}$

We assume that we gained the lower bound for our conditional constraints from statistics over a reasonable amount of pictures form the athletics domain. With an upper bound of 1.0 we wish

to express the default knowledge e.g. "If some hasPart.Bar is evident than this is a HighJump Event". More specific constraints having further evidences gain a higher lower bound probability and therefore their uncertainty interval becomes smaller.

To use the probabilistic lexicographic reasoning services we need to compute the z-partition of $\mathcal{P}_g$. For the example we would have to set up four linear programs to compute part D_0 and one for D_1. The z-partition of our PTBox contains the first three conditional constraints in part 0 and the last, more specific one in the 1 part. The conflicting constraints are one and four.

We now want to use our generic probabilistic terminology to determine the probability intervals of pictures showing specific jumping events. From a picture as shown in Fig. 2.1 image analysis and postprocessing might produce the following PAbox axioms :

$(\exists hasPart.Bar|\top)_{image1}[0.8, 1.0]$
$(\exists hasPart.Pole|\top)_{image1}[0.9, 1.0]$

Here the upper bound is set to 1.0 to express the optimism that the image extraction process identified the right concept. The lower bound is set to the probability provided by the image extraction process denoting the confidence in the identified concept . Notice that a concept which has been detected with a low confidence results in a larger interval of uncertainty.

We also investigated the consequences of a change of confidence in a detected concept with respect to probabilistic instance checking. The following values for $(PoleVault|\top)_{image1}[?, ?]$ and $(HighJump|\top)_{image1}[?, ?]$ have been computed while varying the lower bound of $(\exists hasPart.Pole|\top)_{image1}[l, 1.0]$.

l	0.0-0.4	0.5	0.6	0.7	0.8	0.9	1.0
HJ-u	0.68	0.65	0.58	0.51	0.44	0.37	0.3
HJ-l	0.32	0.32	0.32	0.32	0.32	0.0	0.0
PV-u	0.68	0.68	0.68	0.68	0.68	1.0	1.0
PV-l	0.32	0.35	0.42	0.49	0.56	0.63	0.7

7.6 Average-Case Analysis

The ContraBovemRufum Prototype (`http://www.sts.tu-harburg.de/%7Et.naeth/#software`) is an extension to RacerPro for first experiments with probabilistic approaches in the Boemie project. After a successful implementation of the algorithm proposed in [12] a further objective of this section is the analysis of its average-case behaviour.

Already during the implementation of the class TightLexEntailment it became obvious, that the algorithm in [12] will perform in the average case like $O(2^n)$, where n is the number of conditional constraints within the ith part of the z-Partition.

In order to explain this conclusion, one has to take a closer look at the algorithm. The algorithm may be divided into three parts:

Part 1 tests, if the evidence concept is satisfiable against the TBox($\mathcal{T}_g$) and a set $\mathcal{F}$ of all ABox and PABox axioms bound to a certain individual. $\mathcal{F}$ is only relevant, if probabilistic concept membership is computed.

Part 2 computes lexicographic minimal sets of conditional constraints that are not in conflict with the verified evidence.

Part 3 computes the tightest entailed bounds using the previously computed sets and the conclusion concept.

The reason for poor average case performance is within the second part. Here the computation of the power set for each part $\mathcal{D}_i$ of the z-partition is required in order to iterate through all possible subsets $\mathcal{G}$. A power set has 2^n elements (see proof in [40]) and the algorithm visits all of them. Therefore the average complexity is determined to be $O(2^n)$ and it has been shown that the algorithm is intractable.

Here are some ideas on how to modify the algorithm to improve its average case performance:

As the first idea "lazy power set computation" is introduced. By this is meant, that one starts with $\mathcal{G}$ as the set containing all elements and then with all subsets, where we have $(n-1)$ elements and so on. The cardinality of these sets of subsets is given by $\binom{n}{n-i}$, where n is the number of elements and i the number of omitted elements.

If a set of subsets is found, where some sets are satisfiable, the process may be stopped since we are interested only in those kind of sets which satisfy as many conditional constraints as possible. Still for the worst case all subsets down to the level where the subsets only contain one element have to be computed. But on average the algorithm has to visit less subsets $\mathcal{G}$ than the implemented algorithm.

The second idea is to combine "lazy power set computation" with binary search. This works as follows:

We start with the set of subsets containing $\binom{n}{n-\lceil \frac{n}{2} \rceil}$ and test its elements for satisfiability. If at least one set is satisfied, the search continues within the upper half between n and $\lceil \frac{n}{2} \rceil$; otherwise the search continues in the lower half in the same manner.

These two ideas have been already applied to the algorithm in [22].

A third idea is to introduce some heuristic method before performing binary search in order to reduce the search space. For example we could pick a random subset $\mathcal{G}$ of $\mathcal{D}_j$ and test it for satisfiability. If $\mathcal{G}$ is indeed satsfiable the cardinality of $\mathcal{G}$ is set as new lower bound for the binary search. This makes sense because computing the satisfiability of a random subset is relativly cheap compared to computing set of subsets with $n-i$ elements.

All ideas presented are definitely fields of interest for future research. The ContraBovemRufum Prototype as an extension to RacerPro is prepared for easy integration of these ideas.

7.7 Summary

A prototypical implementation of the inference algorithm for probabilistic description logics proposed by Giugno and Lukasiewicz in [12] has been realised. Afterwards the average case behaviour of the implemented algorithm was evaluated analytically.

The ContraBovemRufum Prototype may be seen as a basic software framework that is capable of easily integrating further description logic reasoners , linear program solver and probabilistic entailment algorithms. Within the scope of this section the description logic reasoner RacerPro was integrated together with the linear program solver from OpsResearch in order to provide the new probabilistic reasoning services *probabilistic subsumption* and *probabilistic instance checking*.

For determining the probabilistic lexicographic entailment, papers [13, 12] have been discussed and the proposed algorithm has been implemented.

As a freebie the prototype is able to handle tight probabilistic logical entailment. This entailment does not allow exceptions in the Probabilistic Knowledge Base.

The average case analysis concluded analytically that the implemented algorithm is intractable. But that does not mean that there is no room for improvement of the average case behaviour, since the algorithm uses a brute force approach to compute the lexicographic minimal set.

Here some ideas for further research and development of the implemented ContraBovemRufum System are given:

- Integration of further solvers, description logic reasoners and of general description logic interfaces DIG and DIG 2.0, enabling a broader application of the system and selection of the user-preferred subsystems.
- Research and development of new tractable algorithms by applying different search strategies to determine lexicographic minimal sets.
- Development of an analysis package to evaluate new algorithms.
- Optimisation of the new algorithms by threaded computation of the variables y_r, of the parts of the z-partition and of lexicographic minimal sets.
- Improvement of the management functionality for Probabilistic Knowledge Bases by implementation of a PKB Broker, which allows to save a PKB using XML and convert it from one reasoner language to another.
- Research and development of applications which can be realised on top of the ContraBovemRufum System

Chapter 8

Conclusion

In order to meet the requirements of the Boemie project, the reasoning engine has been substantially extended. The abduction facility is the most important extension. It is necessary to also support a preference measure in the abduction process in order to avoid unintented high-level interpretations. The implementation could be extended to also support the event recognition rules in the abduction process. In addition, rather than supporting event recognition in a backward-chaining way (in order to detect an event you have to iterate and query over all possible events), it would advantageous to support a forward-chaining style of event recognition (no queries necessary).

An initial implementation of nonstandard inferences are provided for moderately expressive description logics. Based on future research on learning in Boemie, the implementation will be extended to support more expressive logics.

For the probabilistic component, an initial implementation is provided. However, the average-case complexity of the reasoning algorithms is exponential. Thus, the current version can serve as an initial tool for studying how uncertainty can be modeled and how uncertain knowledge can be used in Boemie. It is currently unclear how uncertainty can be considered in the abduction process (e.g., in the preference measure mentioned above).

Parts of this report have been published in workshops and conferences. Namely, Chapter 1 is published in [10] and [27]. The evaluation of the abduction framework is presented in [35]. The event recognition facility is described in [27]. The triple store reasoning facilitiy was presented at SemTech 07. The probabilistic extension to Racer is described in detail in [28].

Bibliography

[1] Fahiem Bacchus. *Representing and reasoning with probabilistic knowledge: a logical approach to probabilities*. MIT Press, Cambridge, MA, USA, 1990.

[2] J. E. Beasley, editor. *Advances in linear and integer programming*. Oxford University Press, Inc., New York, NY, USA, 1996.

[3] Rachel A. Bourne and Simon Parsons. Connecting lexicographic with maximum entropy entailment. In *ESCQARU*, pages 80–91, 1999.

[4] S. Colucci, T. Di Noia, E. Di Sciascio, M. Mongiello, and F. M. Donini. Concept abduction and contraction for semantic-based discovery of matches and negotiation spaces in an emarketplace. In *ICEC '04: Proceedings of the 6th international conference on Electronic commerce*, pages 41–50, 2004.

[5] George Dantzig. George Dantzig Memorial Site. Institute of Operations Research and Management Science, 2005. This is an electronic document retrieved from `http://www2.informs.org/History/dantzig/index.htm`. Date retrieved: January 22, 2007.

[6] George B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In Tjalling C. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. John Wiley & Sons, Inc., New York, 1951.

[7] E. Di Sciascio, F.M. Donini, and M. Mongiello. A description logic for image retrieval. In *Proceedings of the 6th Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, number 1792 in Lecture Notes in Computer Science, pages 13–24. Springer, 1999.

[8] Zhongli Ding and Yun Peng. A probabilistic extension to ontology language owl. *hicss*, 04:40111a, 2004.

[9] C. Elsenbroich, O. Kutz, and U. Sattler. A case for abductive reasoning over ontologies. In *Proc. OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11*, 2006.

[10] S. Espinosa, A. Kaya, S. Melzer, R. Möller, and M. Wessel. Multimedia Interpretation as Abduction. In *Proc. DL-2007: International Workshop on Description Logics*, 2007.

[11] Robert Fourer, David M. Gay, and Brian W. Kerninghan. *AMPL - A Modeling Language for Mathematical Programming*. Thomson Books/Cole, 2003.

[12] Rosalba Giugno and Thomas Lukasiewicz. P-$\mathcal{SHOQ}(\mathbf{D})$: A probabilistic extension of $\mathcal{SHOQ}(\mathbf{D})$ for probabilistic ontologies in the semantic web. Technical Report INFSYS RR-1843-02-06, TU Wien, 2002.

[13] Rosalba Giugno and Thomas Lukasiewicz. P-$\mathcal{SHOQ}(\mathbf{D})$: A probabilistic extension of $\mathcal{SHOQ}(\mathbf{D})$ for probabilistic ontologies in the semantic web. In *JELIA*, pages 86–97, 2002.

[14] Rolf Haenni. Towards a unifying theory of logical and probabilistic reasoning. In *ISIPTA*, pages 193–202, 2005.

[15] Theodore Hailperin. *Boole's Logic and Probability.* Elsevier Science Publishers B.V., second edition edition, 1986.

[16] J. R. Hobbs, M. Stickel, D. Appelt, and P. Martin. Interpretation as abduction. *Artificial Intelligence*, 63:69–142, 1993.

[17] Manfred Jaeger. Probabilistic reasoning in terminological logics. In *KR*, pages 305–316, 1994.

[18] A. Kakas and M. Denecker. Abduction in logic programming. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond. Part I*, number 2407 in LNAI, pages 402–436. Springer, 2002.

[19] Daphne Koller, Alon Y. Levy, and Avi Pfeffer. P-classic: A tractable probabilistic description logic. In *AAAI/IAAI*, pages 390–397, 1997.

[20] Daniel Lehmann. Another perspective on default reasoning. *Annals of Mathematics and Artificial Intelligence*, 15:61–82, 1995.

[21] T. Lukasiewicz. Probabilistic logic programming under inheritance with overriding, 2001.

[22] T. Lukasiewicz. Probabilistic description logics for the semantic web. Technical Report INFSYS RR-1843-06-05, TU Wien, 2006.

[23] Thomas Lukasiewicz. Probabilistic default reasoning with conditional constraints. *CoRR*, cs.AI/0003023, 2000.

[24] John McCarthy. Circumscription - a form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980.

[25] Drew V. McDermott and Jon Doyle. Non-monotonic logic i. *Artif. Intell.*, 13(1-2):41–72, 1980.

[26] R. Möller, V. Haarslev, and B. Neumann. Semantics-based information retrieval. In *Proc. IT&KNOWS-98: International Conference on Information Technology and Knowledge Systems, 31. August- 4. September, Vienna, Budapest*, pages 49–6, 1998.

[27] R. Möller and B. Neumann. Ontology-based reasoning techniques for multimedia interpretation and retrieval. In *Semantic Multimedia and Ontologies : Theory and Applications.* 2007. To appear.

[28] Tobias H. Naeth. Analysis of the average-case behavior of an inference algorithm for probabilistic description logics. Master thesis, TU Hamburg-Harburg, February 2007.

[29] B. Neumann and R. Möller. On Scene Interpretation with Description Logics. In H.I. Christensen and H.-H. Nagel, editors, *Cognitive Vision Systems: Sampling the Spectrum of Approaches*, number 3948 in LNCS, pages 247–278. Springer, 2006.

[30] Bernd Neumann. Retrieving events from geometrical descriptions of time-varying scenes. In J.W. Schmidt and Costantino Thanos, editors, *Foundations of Knowledge Base Management – Contributions from Logic, Databases, and Artificial Intelligence*, page 443. Springer, 1985.

[31] Bernd Neumann and Hans-Joachim Novak. Event models for recognition and natural language description of events in real-world image sequences. In *Proc. of the 8th Int. Joint Conf. on Artificial Intelligence (IJCAI'83)*, pages 724–726, 1983.

[32] Manfred Padberg. *Linear Optimization and Extensions*. Springer, 1995.

[33] Judea Pearl. *Probabilistic Resoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., 1988.

[34] Judea Pearl. System z: A natural ordering of defaults with tractable applications to nonmonotonic reasoning. In *Proceedings of the 3rd Conference on Theoretical Aspects of Reasoning about Knowledge*, pages 121–135, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[35] S. Espinosa Peraldi, A. Kaya, S. Melzer, R. Möller, and M. Wessel. Towards a media interpretation framework for the semantic web. *The 2007 IEEE/WIC/ACM International Conference on Web Intelligence (WI'07)*, 2007.

[36] Raymond Reiter. A logic for default reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980.

[37] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[38] Murray Shanahan. Perception as Abduction: Turning Sensor Data Into Meaningful Representation. *Cognitive Science*, 29(1):103–134, 2005.

[39] Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

[40] H. F. Mattson, Jr. *Discrete Mathematics with Applications*, chapter 3, pages 120–121. John Wiley & Sons, Inc., 193.