KIT REPORT 124 [1]

The FLEX System

J. Joachim Quantz, Guido Dunker,
Frank Bergmann, Ivonne Kellner

TECHNISCHE UNIVERSITÄT BERLIN
PROJEKT KIT, FR 5–12
FRANKLINSTR. 28/29, W–1000 BERLIN 10

e-mail: flex@cs.tu-berlin.de

December 1995

[1]This work was funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verbmobil Project under Grant 01 IV 101Q8. The responsibility for the contents of this study lies with the authors.

Abstract

This report describes the Description Logic (DL) system FLEX. It consists of a brief overview over the field of Description Logics in general and the characteristics of FLEX, a tutorial for the FLEX system, a brief description of the inference algorithms, and an appendix containing a syntax overview, the formal semantics, a reference manual, and an installation guide.

In a sense, the FLEX system can be seen as an extension of the DL system BACK. The main differences are that FLEX supports full disjunction and negation, weighted defaults, situated object descriptions, term-valued features, and flexible inference strategies. On the other hand, FLEX does not support some of the functionality provided by the BACK system, such as revision, for example.

The FLEX system is developed in the project KIT-VM11, which is part of the VERBMOBIL project, a project concerned with face-to-face dialogue interpreting funded by the German Ministry of Education, Science, Research and Technology. Our main criteria for designing the FLEX system are therefore based on requirements arising from the application of semantic disambiguation in Machine Translation.

Contents

Chapter 1

Introduction

In this report we describe the Description Logic (DL) system FLEX. Our main goal is to explain the functionality of FLEX to users of the system. Chapter 3 therefore contains a *tutorial* explaining the most important features of the FLEX system by means of a small modeling example. The appendix contains an *installation guide* (Chapter A), a *syntax overview* (Chapter B), a *formal semantics* (Chapter C) and a *reference manual* (Chapter D), in which all language constructs supported by FLEX are described in detail.

In addition to this rather application-oriented presentation, we also include a chapter describing the field of DL in general and the characteristics of FLEX in particular (Chapter 2), as well as a chapter on implementation issues (Chapter 4). Finally, Chapter 5 contains an overview over future developments envisaged for FLEX.

The FLEX system is developed in the project KIT-VM11, which is part of the VERBMOBIL project, a project concerned with face-to-face dialogue interpreting funded by the German Ministry of Education, Science, Research and Technology. Our main criteria for designing the FLEX system are therefore based on requirements arising from the application of semantic disambiguation in Machine Translation. We would like to thank NManfred Gehrke from Siemens and the other members of KIT-VM11, Uwe Küssner, Juliana Lagunov, Nina Ruge, Birte Schmitz,and Manfred Stedefor their suggestions concerning the design of FLEX interface language and for their patience in working with unstable versions of FLEX.

Many ideas underlying the design and implementation of the FLEX system are based on discussions and experiences from the BACK project. We also used as much of the the BACK manual [Hoppe et al. 93] as possible. We would therefore like to thank Martin Fischer, Thomas Hoppe, Carsten Kindermann, and Albrecht Schmiedel. Thanks are also due to Bob MacGregor for many helpful discussions on implementation issues in the LOOM system.

Chapter 2

The DL System FLEX

2.1 Description Logics

The Roots of Description Logics

The field of Knowledge Representation (KR) originated in the late 1960's and found its first paradigm in *Semantic Networks* as proposed in [Quillian 68]. Quillian assumed semantic memory to be general memory, underlying such diverse cognitive activities as language understanding and perception. The intimate connection between his work and research in psychology and linguistics is typical for this early phase of KR—it was assigned the task of modeling the human ability to use information for intelligent activities such as natural language understanding, perception, planning, etc. Consequently, psychological experiments were conducted to determine the cognitive adeqacy of the proposed formalisms [Collins, Quillian 69].

The second paradigm of KR was created by Minky's seminal paper on *Frames* [Minsky 75]. Minsky's original goal was to account for the effectiveness of common-sense reasoning in real-world tasks — to a large degree motivated by "human" models borrowed from psychology or text linguistics.

Though not obvious on first sight, there is a close resemblance between Semantic Nets and Frames. A semantic net consists of nodes that are connected by labelled links. Some of these links, the so called ISA-Links indicate specialization, while others stand for other relations holding between the nodes. A key idea of Semantic Nets is the notion of *inheritance*: the links attached to a node are inherited to its specializations. Due to their graphical notation and the inheritance principle, Semantic Nets thus allow a compact representation of the complex dependencies between pieces of information.

The frames correspond to the nodes in a semantic net, the ISA-Link is represented by the subframe relation, and the labelled links are modeled by *slots* and *fillers* (a slot correponds to the labelled link, the filler to the node to which the link points). Note that inheritance from frames to subframes mirrors the inheritance via ISA-Links in Semantic Networks.

Though both representation formats are thus very similar, there are also important differences. Semantic Nets are more appropriate for conveying the general *structure* of the represented information, especially interconnections and dependencies; a frame representation, on the other hand, focuses not so much on the overall structure but on the basic units and the information locally associated with each frame. Note that this difference between both formalisms is not an issue of expressive adequacy but one of structural adequacy. Both Semantic Nets and Frames are ancestors of Description Logics and all three approaches to KR have much in common. There are, however, essential characteristics of DL that distinguish them from their ancestors—the basic difference concerns the attitude towards theoretical foundations and towards the question of what is constitutive for a representation *formalism*.

In the second half of the 1970's representation languages from the area of Semantic Nets, Frames, or Scripts were seriously attacked in a number of papers for their apparent lack of formal rigor (e.g., [Woods 75] and [Hayes 77]). The key issue was the relationship between Knowledge Representation and Formal Logic. Note that Quillian located his semantic memory between natural languages and symbolic logic [Quillian 68, p. 230]. This opinion clearly implies that Semantic Nets are superior to Predicate Logic with respect to expressive adequacy.

One apparent problem we face here is the missing entailment relation for Semantic Nets or Frames—how do we know whether a semantic net contains more information than another? Of course, this boils down to the question of how we know what exactly a semantic net or a frame means?

Hayes answers this question by mapping frame definitions into FOL formulae [Hayes 80], and thus shows that Frames are not superior to FOL with respect to expressive adequacy. Two remarks seem in order here. First, the above argumentation is, at least to a certain degree, circular—we do not know what formulae in a formalism mean unless we define a notion of entailment; given such a notion of entailment we have a logic; thus all representation formalisms are logics. Second, if representation formalisms are reducible to FOL it does not mean that they are dispensable. It just means that they are not superior to FOL with respect to expressive adequacy, but they stil can be with respect to structural or computational adequacy. Reducibility to FOL is rather an advantage because it guarantees applicability of the tools for theoretical investigations developed in Formal Logic. To draw an analogy to Programming Languages—nobody would discard a programming language just because it can be compiled into an already existing language. On the contrary, the new language has to be compiled into Machine Code ultimately, and it is confined to the limits of theoretical computability as expressed by Turing Machines, the Lambda Calculus, etc.

Brachman endorsed the logic-oriented view on Knowledge Representation in his early papers on Semantic Nets [Brachman 77, Brachman 79]. Instead of formalizing Semantic Nets in terms of FOL, however, he examined in detail, what the constructs used in them were supposed to represent. There are two important results of his investigations: for one thing, Brachman proposes to distinguish several levels in the discussion of knowledge representation systems, namely the *implementational*, the *logical*, the *epistemological*, the *conceptual*, and the *linguistic* level [Brachman 79, p. 28ff]. In addition, he proposes KL-ONE as a formalism on the epistemological level and presents its basic elements or *epistemological primitives*.

An overview over the basic features of the KL-ONE formalism circulated in the beginning of the 80's and was finally published in [Brachman, Schmolze 85]. In the following years several DL systems have been developed incorporating different dialects but similar with respect to the underlying representation philosophy. The respective formalisms and systems were called KL-ONE alike systems, term subsumption systems, concept logics, terminological logics, and description logics.

Theoretical Investigations

Though Brachman mentioned the importance of a formal semantics in his papers, it took some years before Description Logics were thoroughly investigated from a theoretical point of view. Schmolze and Israel presented a formal semantics for KL-ONE in [Schmolze, Israel 83]. One year later, Brachman and Levesque raised the question of tractability of KR formalisms [Brachman, Levesque 84]. In the late 1980's several results were obtained concerning the tractability of different DLs. In order to understand these results we now have to take a closer look on the theoretical foundations of DL.

In DL one typically distinguishes between *terms* and *objects* as basic language entities from which three kinds of formulae can be formed: *definitions*, *descriptions*, and *rules* (see the sample model on page 6 below). A definition has the form $\mathrm{t}_n \doteq \mathrm{t}$ and expresses the fact that the name t_n is used as an abbreviation for the term t. A list of such definitions is often called *terminology* (hence also the name Terminological Logics). All DL dialects provide two types of terms, namely *concepts* (unary predicates) and *roles* (binary predicates), but they differ with respect to the term-forming operators they contain. Common concept-forming operators are: conjunction (c_1 **and** c_2), disjunction (c_1 **or** c_2), and negation (**not**(c)), as well as quantified restrictions [Quantz 92a] such as value restrictions ($\forall$r:c), which stipulate that all fillers for a role r must be of type c, or number restrictions ($\geq n$ r:c or $\leq n$ r:c), stipulating that there are at least or at most n role-fillers of type c for r. Role-forming operators are, besides conjunction, disjunction, and negation, role composition ($\mathrm{r}_1.\mathrm{r}_2$), transitive closure ($\mathrm{r}^+$), inverse roles ($\mathrm{r}^-$) and domain or range restrictions ($_\mathrm{c}|\mathrm{r}$ or $\mathrm{r}|_\mathrm{c}$). In a description, an object is described as being an instance of a concept (o :: c), or as being related to another object by a role (o_1 :: r:o_2). Rules have the form $c_1 \Rightarrow c_2$ and stipulate that each instance of the concept c_1 is also an instance of the concept c_2.

A formal syntax and semantics for DL is given in Chapter C. Note that DL are subsets of First-Order Logic (with Equality), which can be shown easily by specifying translation functions from DL formulae into FOL formulae [Schmolze, Israel 83, Royer, Quantz 92].

To summarize these theoretical issues, we can say that DLs are characterized by a particular stance towards the essentials of KR-formalisms—in order to call something a KR formalism

1. it has to be a formal language in the sense that there is a formal specification of its syntax;

2. it has to have a formal semantics which defines an entailment relation on formulae;

3. there have to be (efficient) algorithms computing entailment between formulae.

Implemented Systems and Applications

From the beginning on, research in DL was praxis-oriented in the sense that the development of DL systems and their use in applications was one of the primary interests. In the first half of the 1980's several systems were developed that might be called in retrospection *first-generation* DL systems. These systems include KL-ONE [Brachman, Schmolze 85], NIKL [Schmolze, Mark 91], KANDOR [Patel-Schneider 84], KL-TWO [Vilain 85], KRYPTON [Brachman et al. 83], MESON [Edelmann, Owsnicki 86], and SB-ONE [Kobsa 89].

In the second half of the 1980's three systems were developed which are still in use, namely BACK, CLASSIC, and LOOM. The LOOM system [MacGregor 91] is being developed at USC/ISI and focuses on the integration of a variety of programming paradigms aiming at a general purpose knowledge representation system. CLASSIC [Brachman et al. 91] is an ongoing AT&T development. Favoring limited expressiveness for the central component it is attempted to keep the system compact and simple so that it potentially fits into a larger, more expressive system. The final goal is the development of a deductive, object-oriented database manager. BACK [Hoppe et al. 93] is intended to serve as the kernel representation system of AIMS (Advanced Information Management System), in which tools for semantic modeling, defining schemata, manipulating data, and querying, will be replaced by a single high-level description interface. To avoid a "tool-box-like" approach, all interaction with the information repository occurs through a uniform knowledge representation system, namely BACK, which thus acts as a mediating layer between the domain-oriented description level and the persistency level. The cited systems share the notion of DL knowledge representation as being the appropriate basis for expressive and efficient information systems [Patel-Schneider 87]. In contrast to the systems of the first generation, these *second generation* DL systems are full-fledged systems developed in long-term projects and used in various applications. [1]

The systems of the second generation take an explicit stance to the problem that subsumption determination is at least NP-hard or even undecidable for sufficiently expressive languages: CLASSIC offers a very restricted DL and almost complete inference algorithms, whereas LOOM provides a very expressive language but is incomplete in many respects. Recently, the KRIS system has been developed, which uses tableaux-based algorithms and provides complete algorithms for a very expressive DL [Baader et al. 92]. KRIS might thus be the first representative of a third generation of DL systems, though there are not yet enough experiences with realistic applications to judge the adequacy of this new approach.[2]

In order to get a better understanding of these systems let us take a look at the modeling scenario assumed for applications. An application in DL is basically a *domain model*, i.e. a list of definitions, rules, and descriptions (a set of DL-formulae Γ). Consider the highly simplified domain model below, whose net representation is shown in Figure 2.1. One role and five concepts are defined, out of which four are primitive (only necessary, but no sufficient conditions are given). Furthermore, the model contains one rule and four object descriptions.

product	:<	anything
chemical product	:<	product
biological product	:<	product & not(chemical product)
company	:<	some(produces,product)
produces	:<	domain(company)
chemical company	:=	company &
		all(produces,chemical product)
some(produces,chemical product)	=>	high risk company
toxipharm	::	chemical product
biograin	::	biological product
chemoplant	::	chemical company
toxiplant	::	atmost(1,produces) &
		produces:toxipharm

[1] In addition to these general purpose DL systems, some special-purpose systems, such as QUERELLE [Decio et al. 91], K-REP, or KRAPFEN were developped in the second half of the 1980's.

[2] The missing constructiveness of the refutation oriented tableaux algorithms (see Section 4.1) leads to problems with respect to object recognition and retrieval (see [Kindermann 95]).

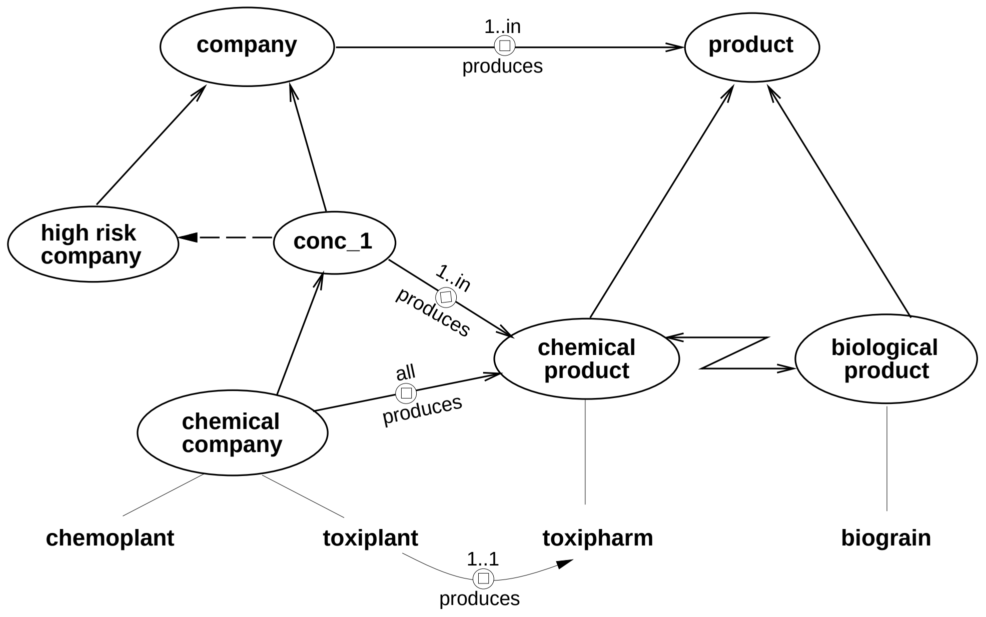


Figure 2.1: The net representation of the sample domain. ‘conc_1’ is the concept some(produces,chemical product).

In DL, such a model is regarded as a set of formulae Γ. Given the formal semantics of a DL, such a set of formulae will entail other formulae, i.e., there is an entailment relation $\Gamma \models \gamma$. Now the service provided by DL systems is basically to answer queries whether some formula γ is entailed by a model Γ. The following list contains examples for the types of queries that can be answered by a DL system:

- $\Gamma \models \mathrm{t}_1 \sqsubseteq \mathrm{t}_2$
  Is a term t_1 more specific than a term t_2, i.e., is t_1 *subsumed* by t_2? In the sample model, the concept ‘chemical company’ is subsumed by ‘high risk company’, i.e., every chemical company is a high risk company.

- $\Gamma \models \mathrm{t}_1$ **and** $\mathrm{t}_2 \sqsubseteq$ **nothing**
  Are two terms t_1 and t_2 incompatible or disjoint? In the sample model, the concepts ‘chemical product’ and ‘biological product’ are disjoint, i.e., no object can be both a chemical and a biological product.

- $\Gamma \models \mathrm{o} :: \mathrm{c}$
  Is an object o an instance of concept c (object classification)? In the sample model, ‘toxiplant’ is recognized as a ‘chemical company’.

- $\Gamma \models o_1 :: r{:}o_2$
  Are two objects o_1,o_2 related by a role r, i.e., is o_2 a role-filler for r at o_1? In the sample model, 'toxipharm' is a role-filler for the role 'produces' at 'toxiplant'.

- $\Gamma \models X :: c$
  Which objects are instances of a concept c (retrieval)? In the sample model, 'chemoplant' and 'toxiplant' are retrieved as instances of the concept 'high risk plant'.

- $\Gamma \cup \{\alpha\} \models \perp$
  Is a description α inconsistent with the model (consistency check)? The description 'chemoplant :: produces:biograin' is inconsistent, wrt the sample model, i.e., 'biograin' cannot be produced by 'chemoplant'.

This very general scenario can be refined by considering generic application tasks such as *information retrieval*, *diagnosis*, or *configuration*.

Extending Description Logics

Research in DL was originally concerned with the set of epistemological primitives presented in [Brachman, Schmolze 85], but the interest in extending DL to integrate results from other fields of AI increased considerably in the last years. The integration of rule-based reasoning into DL was suggested very early in [Owsnicki-Klewe 88]. It turned out, however, that the so-called implication-links or rules which have been implemented in BACK, CLASSIC, and LOOM, differ from the rules of rule-based system with respect to their logical, monotonic character and thus function more as constraints then as production rules [Schild 89]. As a consequence, no conflict resolution strategies have to be devised. Thus the integration of rules kept very close to the original paradigm of DL.

The epistemological primitives of KL-ONE were meant to be ontologically neutral, i.e., it should be possible to use them for the definition of concepts belonging to arbitrary ontological categories. It became quickly obvious, however, that certain ontological areas need additional term-forming operators which specifically reflect the ontological structure of these areas. Schmiedel proposed an integration of DL, Shoham's temporal logic, and Allen's interval calculus to support the modeling of temporal knowledge in a DL system [Schmiedel 90]. Schmiedel's ideas were taken up by Schild, who developed a restricted tense-logical extension of DL, which constituted an optimal tradeoff between expressiveness and computational complexity [Schild 91]. An integration of this tense-logical extension into the BACK system is described in [Fischer 92].

Another topic of discussion concerned the integration of defeasible knowledge into DL. In the traditional framework of DL all information is regarded as strict. This is obviously a restriction much too strong for most domain: most of our knowledge is expressable only as rules that allow for exceptions or as rules with a certain degree of reliability. In order to capture this kind of information the integration of probabilistic rules [Heinsohn, Owsnicki-Klewe 88], [Heinsohn 91] and of defaults [Baader, Hollunder 92], [Quantz, Royer 92], [Baader, Hollunder 93] was suggested.

Other envisaged extensions of DL include the integration of generalized quantifiers [Quantz 92a], of second-order constructs [Quantz 92b], of part-whole relations and collective entities in general [Franconi 92], of epistemic operators [Donini et al. 92], and of test/compute functions [Kortüm 93].

Some of these extensions have been integrated in the FLEX system, namely weighted defaults and test-compute functions. An integration of epistemic operators is envisaged for the future.

2.2 Characteristics of FLEX

Though the FLEX system follows the general DL paradigm it possesses a number of characteristics not shared by all DL systems:

1. an *expressive term language* comprising qualifying number restrictions, role inversion and composition, role value maps, disjunction and negation;

2. *term-valued features*, which behave similar to features having complex-type values in Unification Grammars;

3. *situated descriptions* which allow the representation of alternative states of affair and form the basis for default reasoning;

4. *weighted defaults* as proposed in [Quantz 93] and formally investigated in [Quantz, Suska 94];

5. *flexible inference strategies.*

In the following we will briefly sketch these characteristics. If you are not familiar with DL, you will probably have difficulties in understanding some of the (technical) remarks. The tutorial in the next chapter will explain in detail how these characteristics of FLEX are to be used in an application.

Types and Objects

One major difference between DL and the feature logics used in Unification Grammars (e.g. [Carpenter 92]) is the fact that the former distinguish between objects and types, whereas the latter only deal with types. This is reflected in the respective syntactic formats by the fact that DL provide two different constructs for constraining role fillers, namely r:o, on the one hand, and **all**(r,c), **some**(r,c), or **the**(r,c) on the other hand. UG only provide the constructor f:v, where 'v' can be an atomic value or a complex feature structure. It should be noted that the object level of DL, which is "missing" in UG, allows a straightforward persistent storage of objects and their types. Furthermore, it is the basis for the integration of *epistemic operators* and *weighted defaults* into DL.

There is, however, one disadvantage of the distinction between types and objects in DL. In standard DL only objects can be fillers of roles. But certain features, e.g. 'has_topic', rather are meant to take concepts as role fillers. Furthermore we would expect has_topic:c_1 to be more specific than has_topic:c_2 if c_1 is subsumed by c_2. We could model this by writing **the**(has_topic,c_1) instead of has_topic:c_1, but this would imply that we have an object filling the role 'has_topic' which is of type c_1.

A correct treatment of this problem would involve a higher-order extension to DL, allowing relations whose arguments can be concepts or roles. Instead of providing such an extension, we integrated term-valued features into the FLEX system. It should be noted that these features "solve" the problem on a syntactic level only, i.e. f_c:c will behave like **the**(f_c,c), except that no terms like **atleast**(n,f_c) or f_c:o are allowed.

The semantics for f_c and f_r will yield the following subsumptions:

$$\begin{array}{rcl} & \rightharpoondown & f_t{:}t_1 \sqcap f_t{:}t_2 \doteq f_t : (t_1 \sqcap t_2) \\ & \rightharpoondown & f_t : \bot \doteq \bot \\ t_1 \sqsubseteq t_2 & \rightharpoondown & f_t{:}t_1 \sqsubseteq f_t{:}t_2 \end{array}$$

Given this basic functionality there is one further extensions allowing to make use of term-valued features. FLEX contains a special construct relating term-valued features to the "normal" features as proposed in [Quantz 92b]:

$$o_1 :: f_c{:}c \sqcap f{:}o_2 \sqcap f{\in}f_c \text{ in } s \quad \rightharpoondown \quad o_2 :: c \text{ in } s$$

This construct can be used to express the constraint that the filler of a role is an instance of the filler of a term-valued feature.

Situated Descriptions

Situated descriptions can be used to represent alternative states of affair, to model backtracking, and to perform reasoning by cases. The basic idea is to describe objects *relative* to a *situation*. Whereas standard DL support only a single ABox (i.e. set of object descriptions), FLEX thus allows partitioning of the ABox into several situations. This is expressed by saying

$$\text{o} \quad :: \quad \text{c in s.}$$

instead of

$$\text{o} \quad :: \quad \text{c.}$$

Note that the latter tell is also accepted by the system and is interpreted as being a description holding in the built-in situation ‘initial’.

Note that situations are not necessarily unrelated but can stand in the extension relation. Intuitively, a situation s_2 extends s_1 if all object descriptions valid in s_1 are also valid in s_2.

Assume that we have the following description:

$$\text{o} \quad :: \quad (\text{f:v}_1 \sqcup \text{f:v}_2) \sqcap \text{c in s}_1.$$

we can then extend the situation s_1 by situations s_2 and s_3:

$$\begin{array}{ccc} s_1 & <<= & s_2. \\ s_1 & <<= & s_3. \end{array}$$

We can now use the situations s_2 and s_3 to perform reasoning by cases:

$$\begin{array}{ccl} \text{o} & :: & \text{f:v}_1 \text{ in s}_2. \\ \text{o} & :: & \text{f:v}_2 \text{ in s}_3. \end{array}$$

All formulae valid both in s_2 and in s_3 are also valid in s_1.

In general, situated descriptions allow the representation of alternative states of affairs and can also be used to realize backtracking. Suppose you are incrementally describing an object, e.g. in a configuration application. Using situated descriptions allows you to jump back to an intermediate state simply by using the respective situation:

$$\begin{array}{ccl} \text{o} & :: & \text{c}_1 \text{ in s}_1. \\ s_1 & <<= & s_2. \\ \text{o} & :: & \text{c}_2 \text{ in s}_2. \\ s_2 & <<= & s_3. \\ \text{o} & :: & \text{c}_3 \text{ in s}_3. \end{array}$$

If it turns out that the last tell was not appropriate, you can create another extension of s_2 to continue:

$$\begin{array}{ccl} s_2 & <<= & s_4. \\ \text{o} & :: & \text{c}_4 \text{ in s}_4. \end{array}$$

In order to give a formal semantics for situated descriptions we have to change the nature of interpretation functions. Instead of mapping concepts into subsets of D, they now map pairs of concepts and situations into subsets of D. Thus the interpretation of a concept can vary from one situation to another. The constraints on interpretation functions then look like:

$$[\![\mathrm{t}_1 \;\mathbf{and}\; \mathrm{t}_2, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \quad = \quad [\![\mathrm{t}_1, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \cap [\![\mathrm{t}_2, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}}$$

and the definition of satisfaction of situated descriptions is rewritten as

$$M \models \mathrm{o} :: \mathrm{c\ in\ s} \quad \text{iff} \quad [\![\mathrm{o}]\!]^{\mathcal{I}} \in [\![\mathrm{c}, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}}$$

Finally, we need a formal semantics of the notion of *extensions* of a situation. What is obviously intended by this notion is that if $\mathrm{s}_1 \leq \mathrm{s}_2$ then o :: c in s_1 implies o :: c in s_2. Note that this is called *persistence* in the context of Situation Semantics [Barwise, Perry 83].

The notion of extensions can be formally defined by using sets of interpretation functions as they are used for the semantics of the epistemic operator **k**:

$$M \models \langle \textit{sit-extension} \rangle \mathrm{s}_1 \mathrm{s}_2 \quad \text{iff} \quad \forall \mathrm{t} [\![\mathbf{k}\mathrm{t}, \mathrm{s}_1]\!]^{\mathcal{I},\mathcal{W}} \subseteq [\![\mathbf{k}\mathrm{t}, \mathrm{s}_2]\!]^{\mathcal{I},\mathcal{W}}$$

Note that this approach to persistency is justified since we only have situated descriptions of individuals. Clearly, we would not want quantified descriptions like ‘all(c_1,c_2) in s_1’ to be persistent (we might learn about an object o not present in s_1 which is a c_1 but not a c_2). Similarly, epistemic formulae behave nonmonotonically and are not persistent. The above definition will therefore be restricted to nonepistemic terms t.

Weighted Defaults

Weighted defaults have been integrated into the FLEX system to support the modeling of rules which allow for exceptions. Syntactically, defaults are similar to strict rules, i.e. they relate two concepts, one being the premise, the other the conclusion:

$$\begin{array}{ccc} \mathrm{c}_1 & => & \mathrm{c}_2. \\ \mathrm{c}_1 & \sim\!\mathrm{n}\!\sim> & \mathrm{c}_3. \end{array}$$

Whereas the strict rule stipulates that any object being a c_1 is also a c_2, the default only stipulates that usually an object which is a c_1 is also a c_3. The weight of the default expresses its strength in cases where conflicts with other defaults arise—the higher the weight the stronger the default.

In Section C we specify a preferential modeltheoretic semantics for weighted defaults. It should be noted that weighted defaults allow to express orderings between multisets of defaults, i.e. weak defaults can accumulate their weights and override a stronger default. This is a rather useful property in many applications [Quantz, Schmitz 94, Schmitz, Quantz 95]. In contrast, most prioritized Nonmonotonic Logics only support orderings between individual defaults, such that a stronger default overrides any number of weaker defaults.

Flexible Inference Strategies

FLEX offers a number of ways to control its inference behavior, which is based on sequent-style inference rules [Royer, Quantz 92, Royer, Quantz 94]. The basic idea is a declarative representation of these inference rules, which allows for most rules to chose between

- an application during normalization;
- an application during subsumption checking;
- no application at all.

Furthermore, the inference behavior can be controlled by switching off propagation rules for roles via the filter construct, and by setting FLEX states.

The possibility to tailor the inference capabilities of FLEX proved rather usefull in our VERBMOBIL application, but it is not yet customized towards the end user. Consequently we will not explain in detail how to control the inference behavior in this report, but we will say somethin about the underlying ideas in Chapter 4.

Chapter 3

Tutorial

In this chapter we give a brief introduction into the use of the FLEX system. Our main purpose is to introduce the functionality provided by the system and to illustrate it with small examples. Note that we assume no prior knowledge of DL in our presentation. We strongly encourage the reader, however, to use an installation of the FLEX system to actually test out the examples presented in the following. Installation of the FLEX system is described in detail in Section A. Here we assume that the system already has been installed. All following examples can be found in the DOCU directory in the file ‘examples’.[1]

As explained in detail in Section 2.1, in DL systems one distinguishes between *concepts*, *roles*, and *objects*. These entities can be used in four different types of formulae:

term introductions	t	:<	t_n.
	t	:=	t_n.
(situated) object descriptions	o	::	c (in s).
rules	c_1	=>	c_2.
weighted defaults	c_1	$\sim$n$\sim>$	c_2.

In the next Section we introduce the basic term language of FLEX, i.e. the operators provided for the modeling of concepts and roles. In doing so, we also illustrate the impact of these operators by considering object descriptions and rules. Section 3.2 then describes more advanced interactions, such as reading and dumping large models from/on files, controling the system behavior through setting of states, and usage of the programming interface. Finally, Section 3.3 explains the use of weighted defaults.

[1] The file ‘examples.model’ can be read in as will be explained in Section 3.2.

3.1 The Basics

In DL one usually distinguishes between a *TBox* and an *ABox*. The TBox contains the definition of the terminology, whereas the ABox contains information about individual objects. Though this distinction is useful for several purposes, we prefer an integrated presentation in the following, i.e. we will freely mix term definitions and object descriptions since this allows a better illustration of the respective impact of the various term-forming operators.

Defining Concepts

Like in all Description Logics, a distinction is made between concepts (unary predicates) and roles (binary predicates). Since roles can be used in the definition of concepts, and vice versa, we do not attempt a strictly separated introduction, however. Before presenting the various concept-forming operators supported by FLEX let us briefly reconsider the general purpose of concept definitions. Basically, concepts are used to describe objects, i.e. concepts *structure* a domain of individuals. There are several respects in which structuring a domain is desirable. The most basic one concerns storing and retrieval of objects—if you define a concept 'book' you can use this concept to

1. inform the system that a particular object is an instance of the concept 'book';

2. let the system retrieve all known instances of the concept 'book'.

Obviously, storing and later retrieving instances of a concept in this simple manner is not very interesting. Things become more interesting, however, if concepts are not only used as atomic labels, but rather are *semantically* related to other concepts. The most basic way of relating concepts in a DL system is by building a *conceptual hierarchy*. The easiest way of building such a hierarchy is to "define" a concept by specifying its super concepts, as illustrated in the following example:

```
            flexinit.
         publication  :<  ctop.
             article  :<  publication.
  conference_article  :<  article.
     journal_article  :<  article.
```

The initial command FLEXINIT is used to inform the system that this is the beginning of a model (its effects will be explained in detail on page 31.) The subsequent tells are called *primitive concept introductions*.

Please note two rather important restrictions wrt such term introductions:

1. There can be *only one definition* for each term name. It is thus not possible to incrementally define concepts or to redefine them. Given the above tells, the additional tell

   conference_article :< publication. `% this tell fails`

   therefore produces an error message indicating that the concept 'conference_article' has already been defined.

2. Terms used in a term introduction must have been introduced before. Thus it is not possible to type in

   biography :< book. `% this tell fails`

   if book has not been introduced before.[2]

Conceptual hierarchies form the basis of *inheritance*, the most fundamental inference capability of DL systems. We can thus ask the system whether a concept *is subsumed* by some other concept, i.e. whether it is more specific. Given the introductions above, the subsumption queries

journal_article ?< article.
journal_article ?< publication.

will succed, whereas the query

publication ?< article. `% this query fails`

will fail.

Describing Objects

In many applications subsumption queries will not be used explictly. It should be noted, however, that subsumption relations provide the basis for object-level reasoning. If the system is told that a particular object is an instance of 'article', it immediately infers that it is also an instance of 'publication':

chinese_room :: article.
chinese_room ?: publication.

Whereas each term is only introduced once, an object can be described incrementally, i.e. there may be several descriptions of the same object. We can thus add the more specific description

[2]In Section 3.2 we explain how the behavior of FLEX can be modified by setting *system states/flags*. When the state 'introduction' is set to 'forward', FLEX automatically introduces all undefined terms used in a definition as primitive terms.

chinese_room :: journal_article.

Whereas no variables are allowed in subsumption queries, instantiation queries can be used to backtrack all instances of a concept, by using a variable instead of an object name:

X ?: article. `% binds X to chinese_room`

Disjointness

An important characteristics of DL systems is that they apply an *open-world semantics* instead of the closed-world semantics underlying PROLOG and standard databases.[3] As a consequence, it must be explicitly modeled whether concepts are to be treated as being disjoint or not. In the above example, 'journal_article' and 'conference_article' are not modeled as disjoint concepts and hence the system will not reject a description in which a publication is both a journal article and a conference article.

```
multi_pub :: journal_article.
multi_pub :: conference_article.
```

The easiest way to represent disjointness between concepts is to use the <> operator:

```
scientific_publication <> fiction.
scientific_publication :< publication.
               fiction :< publication.
```

The <> operator marks 'scientific_publication' and 'fiction' as disjoint concepts. Note that this disjointness declaration has to *precede* the definition of the concepts and that only primitive concepts can be marked as being disjoint.

Given this model, the system will reject a description in which a publication is both a scientific publication and fiction:

non_ex_pub :: scientific_publication and fiction. `% this tell fails`

Note that disjointness of concepts can be checked on the terminological level by checking whether their conjunction is subsumed by CBOT:

scientific_publication and fiction ?< cbot.

Note further that the FLEX system rejects incoherent object descriptions but accepts definitions which yield incoherent concepts:

[3] This difference has consequences wrt negation and monotonicity and has been explicitely addressed in the context of epistemic operators [Donini et al. 92].

```
incoherent_publication  :<  scientific_publication and fiction.
```

The system produces a warning, whenever an incoherent concept is introduced, however.

FLEX also offers the NOT operator for concept negation. The main difference between NOT and <> is that the former negates the whole term whereas the latter only negates the primitive component of a term. We will illustrate this distinction in detail below, when discussing disjunction. For the time being it is sufficient to know that

1. <> does not introduce the complexity that NOT does;

2. <> is appropriate for expressing disjointness between terms.

The operator <> can also be used as a prefix operator taking a list of concepts as arguments. All concepts specified in the list are then marked as being mutually disjoint.

Defined and Primitive Concepts

We will now illustrate the rather important distinction between primitive terms and defined terms. Consider the following definitions:

```
          book  :<  publication.
         novel  :<  book and fiction.
scientific_book :=  book and scientific_publication.
```

Note that 'novel' is introduced as a *primitive* concept, whereas 'scientific_book' is a *defined* concept. The standard explanation of this distinction is that for primitive terms only necessary conditions are specified, whereas for defined terms necessary and sufficient conditions are specified. Thus we know that any 'novel' is a 'book' and 'fiction', i.e. we know necessary conditions about 'novel'.

```
alice  ::  novel.
alice  ?:  book.
alice  ?:  fiction.
```

For 'scientific_book' we also know sufficient conditions, i.e. any object which is an instance of both 'book' and 'scientific_publication' will be inferred to be an instance of 'scientific_book'.

```
origin  ::  book and scientific_publication.
origin  ?:  scientific_book.
```

Objects which are instances of both 'book' and 'fiction', on the other hand, are *not* inferred to be instances of 'novel':

short_stories	::	book and fiction.
short_stories	?:	novel. `% this query fails`

The following subsumption queries illustrate this distinction on the terminological level:

novel	?<	book and fiction.
book and fiction	?<	novel. `% this query fails`
scientific_book	?<	book and scientific_publication.
book and scientific_publication	?<	scientific_book.

Thus the decision wether to model a term as a primitive or as a defined term should be ultimately based on the required inferences. It should be noted that a careful analysis of the required inference is in general the most appropriate basis to choose between modeling alternatives.

Disjunction

In addition to concept conjunction the FLEX system also supports concept *disjunction* and *negation*. The user should be warned, however, that the use of these operators is a notorious source of complexity and can thus have considerable impact on the performance of the system. In many cases, a model containing negation and disjunction can be transformed into a simpler model which yields similar results but does not introduce the complexity caused by disjunction or negation. We have already pointed out that the <> operator is appropriate in this sense for modeling disjointness.

To illustrate the specific contribution of disjunctions in definitions consider the following alternative definitions:

scientific_publication1	<>	fiction1.
scientific_publication1	:<	ctop.
fiction1	:<	ctop.
publication1	:=	scientific_publication1 or fiction1.

Note that the subsumption relations are identical for both alternatives:

scientific_publication	?<	publication.
fiction	?<	publication.
scientific_publication1	?<	publication1.
fiction1	?<	publication1.

The difference concerns *reasoning* by cases, i.e. if a concept is defined as a disjunction and an object is known to be an instance of this concept and to be *not* an instance of all the disjuncts but one, then it must be an instance of this disjunct:

```
ulysses  ::  publication1 and not(scientific_publication1).
ulysses  ?:  fiction1.
```

Note that such an inference is not possible in our original model since it is nowhere specified that a 'publication' is either a 'scientific_publication' or 'fiction'. Given the open-world semantics, there could be, for example another subconcept of 'publication':

```
ulysses  ::  publication and not(scientific_publication).
ulysses  ?:  fiction.        % this query fails
```

Not that reasoning by cases is usually triggered by telling "negative facts". If this does not occur in your application, i.e. if reasoning by cases is not required in your application, you should definitely avoid the use of negation and disjunction in your definitions.

Defining Roles (Domain and Range)

We will now turn our attention towards the definition of roles and the use of roles in concept definitions. Whereas concepts correspond to unary predicates, roles correspond to binary predicates/relations. Consequently, roles can be described by specifying the type of their first argument (DOMAIN) or their second argument (RANGE).

Note that roles are often considered to be secondary compared to concepts, i.e. roles are often seen as properties of concepts or objects and not as independently existing entities. Consider the concept 'publication'. Meaningful properties of publications are 'author', 'title', 'publication_year', etc. We can thus introduce a role

```
has_author  :<  domain(publication).
```

This definition only constrains the domain of 'has_author' but does not contain any restriction wrt its range, i.e. the objects which are allowed as fillers for the role 'has_author'. Moreover, it is possible that a publication has more than one author:

```
chinese_room  ::  has_author:searle.
   principia  ::  has_author:[russell,whitehead].
```

Note that the operator ':' can thus take both an object and a list of objects as second argument.

Note that the domain information specified for the role 'has_author' is sufficient for the system to infer that 'principia' is a publicaiton:

```
principia  ?:  publication.
```

Features

Since many roles are functional, i.e. can have at most one filler per object, FLEX supports the definition of *features*, as exemplified by

```
publication_year   :<   domain(publication) and range(number) and feature.
```

Thus it is not possible to specify two different publication years for a publication:

```
non_ex_pub   ::   publication_year:1985 and
                  publication_year:1987. % this tell fails
```

Another important aspect illustrated by this example is the use of the built-in concept NUMBER. FLEX offers special operators like GT or LE to support the definition of number ranges. The following example illustrates such a definition of a number range and the corresponding consistency check performed on the object level:

```
sixties_publication   :=   publication and the(publication_year,1960..1969).
              do_it   ::   publication_year:1969.
              do_it   ?:   sixties_publication.
         non_ex_pub   ::   sixties_publication and
                           publication_year:1985. % this tell fails
```

Note that this example uses the role-restriction operator THE which will be discussed in more detail now.

Role Restrictions

The operator THE used in the above example takes as first argument a role and as second argument a concept, as do the operators SOME and ALL. The respective meaning of these operators can be informally described as follows:

some(r,c): describes those objects which have a filler for role r which is an instance of c.

all(r,c): describes those objects whose fillers for role r are all instances of c.

the(r,c): describes those objects which have exactly one filler for role r which has to be an instance of c.

The following example illustrates the SOME operator:

```
    famous   :<   ctop.
   russell   ::   famous.
principia    ?:   some(has_author,famous).
```

Furthermore, if we tell the system that all the authors of the Principia are famous, Whitehead immediately becomes famous too:

```
 principia  ::  all(has_author,famous).
whitehead   ?:  famous.
```

We will illustrate the reverse inference, namely that a book has the property that all its authors are famous, below in the context of number restrictions.

Number Restrictions

Another type of role restrictions supported by DL systems are *number restrictions*. The following example contains two definitions which use a minimum and a maximum restriction, respectively:

```
individual_publication  :=  atmost(1,has_author).
     group_publication  :=  atleast(3,has_author).
```

Due to the open world semantics employed in DL systems, FLEX can only infer minimum restrictions but no maximum restrictions from a specification of fillers:

```
epr  ::  has_author:einstein and has_author:podolsky and has_author:rosen.
epr  ?:  group_publication.
epr  ?:  atmost(3,has_author).              % this query fails
```

Thus if inferences involving maximum restrictions are required, the system has to be informed that all role fillers have been specified by explicitly telling a maximum restriction. Note that this also applies to the abstraction of value restrictions:

```
arithmetik  ::  has_author:frege.
     frege  ::  famous.
arithmetik  ?:  individual_publication. % this query fails
arithmetik  ?:  all(has_author,famous). % this query fails
arithmetik  ::  exactly(1,has_author).
arithmetik  ?:  individual_publication and all(has_author,famous).
```

Even though the system does not infer maximum restrictions it checks whether a known maximum restriction is violated. Thus it is not possible to enter an 'individual_publication' having two authors:

```
non_ex_pub  ::  individual_publication and has_author:quine and
                has_author:derrida.   % this tell fails
```

FLEX also supports *qualifying number restrictions*, i.e. number restrictions taking as an additional argument a concept type. Whereas "plain" number restrictions check how many fillers an object has for a certain role, qualifying number restrictions check the number of fillers of a certain type:

```
X  ?:  atleast(2,has_author,famous). % binds X to principia
```

In general, qualifying number restrictions are equivalent to “plain” number restrictions for a range-restricted role:

```
             atleast(2,has_author,famous)  ?<  atleast(2,
                                                has_author and range(famous)).
atleast(2,has_author and range(famous))  ?<  atleast(2,has_author,famous).
```

Before presenting additional operators for role definitions, we will briefly sketch the use of the ONEOF operator.

Extensional Concepts

The concept-building operators introduced so far can be used to *intensionally* describe concepts, i.e. to describe properties shared by their instances. In some cases, however, it is useful to extensionally define concepts, i.e. to explicitly list their instances. This is achieved by using the operator ONEOF, as illustrated by the euro-centric example below:

```
         country  :=  oneof([france,germany, italy,japan,russia,uk,usa]).
european_country  :=  oneof([france,germany,italy,uk]).
european_country  ?<  country.
```

It should be noted that the objects listed in these concept definitions “inherit” the appropriate conceptual information:

```
france  ?:  european_country.
 japan  ?:  european_country. % this query fails
```

Inverse Roles and Role Composition

After the excursion into geopolitics we now return to the presentation of role-forming operators. We have already introduced DOMAIN and RANGE above and will now introduce role inversion and composition.

The operator INV is used to define inverse roles. Whereas the role ‘has_author’ relates publications and authors from the perspective of publications, its inverse ‘has_written’ relates them from the perspective of authors:

```
has_written  :=  inv(has_author).
          X  ?:  has_written:principia.
                 % binds X to russell;whitehead
  principia  ::  book.
book_author  :=  some(has_written,book).
     russel  ?:  book_author.
```

Role composition is useful to model *role chains*. So far we have treated the objects related to publications as atomic. In general, however, most objects are structured, and role chains can be used to exploit this structure. To illustrate this we introduce the following roles:

institution	:<	ctop.
at_institution	:<	range(institution) and feature.
in_country	:<	domain(institution) and range(country) and feature.

Given a publication we might then be interested in the the country in which the institution is located at which the authors work:

from_institution	:=	has_author comp at_institution.
from_country	:=	from_institution comp in_country.

Note that the system can already infer from this definition the domain and the range of the newly defined role:

from_country	?<	domain(publication).
from_country	?<	range(country).

The following example illustrates the most important inference wrt role composition, namely the derivation of role fillers for role chains:

flex	::	has_author:quantz.
quantz	::	at_institution:tub.
tub	::	in_country:germany.
flex	?:	from_country:germany.

Term-Valued Features

The roles we have modeled so far were all used to relate individual objects. Some features, however, seem to be more adequately modeled as relating individual objects with concepts. Consider as an example the topic of a publication, e.g. ‘description logics’, ‘knowledge representation’, or ‘artificial intelligence’. Instead of treating these topics as object, it might be more appropriate to treat them as concepts which makes it possible to order them in a conceptual hierarchy:

topic	:<	ctop.
artificial_intelligence	:<	topic.
knowledge_representation	:<	artificial_intelligence.
description_logics	:<	knowledge_representation.
has_topic	:<	domain(scientific_publication) and term_valued.

Given this hierarchical model we obtain the obvious inferences (both for objects and on the terminological level):

```
                    flex  ::  has_topic:description_logics.
                       X  ?:  has_topic:knowledge_representation.
                                        % binds X to flex
has_topic:description_logics  ?<  has_topic:artificial_intelligence.
```

Note that in principle arbitrary concept or role terms, and not just names, can be specified as fillers for term-valued features.

Situated Descriptions

We will now briefly illustrate *situated descriptions*, i.e. the use of situations in object descriptions. The basic idea is to partition the ABox into different situations, such that an object description may hold in one situation but not in another. Moreover, situations can extend each other, which roughly means that all object descriptions holding in a situation also hold in all situations extending it.

For illustration consider the following example:

```
        language   :<   ctop.
unknown_language   :<   language.
         russian   ::   unknown_language in s1.
              s1  <<=   s2.
         english   ::   unknown_language in s2.
         russian   ?:   unknown_language.
                        % this query fails
         russian   ?:   unknown_language in s1.
         russian   ?:   unknown_language in s2.
         english   ?:   unknown_language.
                        % this query fails
         english   ?:   unknown_language in s1.
                        % this query fails
         english   ?:   unknown_language in s2.
```

Note that object descriptions not containing a situation are taken to be descriptions of the built-in situation INITIAL which all other situations extend. Note further that you can use a variable as situation in situated descriptions and on the right-hand side of an extension statement. In this case FLEX generates a situation name ‘ext_sitN’.

Rules

We end this section by briefly introducing the usage of *rules*:[4]

[4] For the concept ‘ai_conference’ we define a filter which we will need below. We will explain the use of filters in detail in Section 3.2.

```
                      conference  :<  ctop.
                   ai_conference  :<  conference with filter=topic_conf_type.
                   at_conference  :<  domain(conference_article) and
                                      range(conference) and feature.
the(at_conference,ai_conference)  =>  has_topic:artificial_intelligence.
```

Whenever an object is subsumed by the left-hand side of a rule, its right-hand side is added to its description:

```
      ijcai91  ::  ai_conference.
tractable_dl   ::  at_conference:ijcai91.
tractable_dl   ?:  has_topic:artificial_intelligence.
```

3.2 Advanced Interactions

In this section we describe more advanced interactions with FLEX, such as the programming interface, the use of filter, the setting of FLEX states and the reading and dumping of knowledge bases.

The Programming Interface

So far we have used only two types of queries to retrieve information from the knowledge base, namely:

```
term_1  ?<  term_2.
object  ?:  concept.
```

The *programming interface* provides a list of more complex queries which can be used to retrieve specific information about terms and objects (see Figure 3.1).[5]

The basic syntax of the programming interface is as follows:

1. flexget(Entity,Method,Result)
2. flexget(Entity,Method,Options,Result)

where 'Entity' is a concept, a role or an object. Depending on the type of the entity different methods can be used to retrieve information. Figure 3.1 lists these methods as well as their respective options and result.Note that some methods require mandatory arguments.

We can roughly distinguish between predicates providing *structural* and *hierarchical* information. To obtain hierarchical information about concepts or roles, four predicates are available:

[5]It should be noted that from a theoretical point of view, these queries can all be reduced to the simple queries about subsumption and instanceship.

Method	at Entity	Options	Result
all(Role)	conc	box	Conc
all(Role,Sit)	obj	box	Conc
atleast(Role)	conc	box	Integer
atleast(Role,Sit)	obj	box	Integer
atleast(Role,Conc)	conc	box	Integer
atleast(Role,Conc,Sit)	obj	box	Integer
atmost(Role)	conc	box	Integer;in
atmost(Role,Sit)	obj	box	Integer;in
atmost(Role,Conc)	conc	box	Integer;in
atmost(Role,Conc,Sit)	obj	box	Integer;in
concepts(Sit)	obj	box	ListOfConc
dir_subs	conc,role	box,filter	ListOfConc/Role
dir_supers	conc,role	box,filter	ListOfConc/Role
domain	role	—	Conc
equivalents	conc,role	box,filter	ListOfConc/Role
fillers(Role)	conc	box	ListOfObj
fillers(Role,Sit)	obj	box	ListOfObj
help	conc,obj,role	—	ListOfMethods
instances(Sit)	conc	box	ListOfObj
msc(Sit)	obj	filter,box	ListOfConc
range	role	—	Conc
subs	conc,role	box,filter	ListOfConc/Role
supers	conc,role	box,filter	ListOfConc/Role
tvf_filler(Role)	conc	box,filter	Conc/Role
tvf_filler(Role,Sit)	obj	box,filter	Conc/Role

Figure 3.1: The programming interface.

dir_subs retrieves the direct subsumees of a concept or a role;

flexget(book,dir_subs,Result).

```
% binds Result to [novel,scientific_book]
```

dir_supers retrieves the direct subsumers of a concept or a role;

flexget(scientific_book,dir_supers,Result).

```
% binds Result to [scientific_publication,book]
```

subs retrieves all subsumees of a concept or a role;

flexget(article,subs,Result).

```
% binds Result to
[cbot,conference_article,journal_article]
```

supers retrieves all subsumers of a concept or a role.
flexget(novel,supers,Result).
```
% binds Result to [ctop,publication,fiction,book]
```

These predicates can be parameterized by two options:

box can be used to take into account the rules modeled in the IBox or the defaults modeled in the DBox (see Section refsec:dbox;

filter can be used to restrict the result to concepts or roles satisfying the respective filter (see next subsection).

The methods provided for retrieving structural information about entities are different for roles on the one hand and concepts and objects on the other hand. For roles two methods are available:

domain retrieves the domain of a role;
flexget(has_author,domain,Result).
```
% binds Result to publication
```

range retrieves the range of a role.
flexget(publication_year,range,Result).
```
% binds Result to number
```

For concepts and objects the following methods are available. It should be noted that these methods require an argument specifying the situation when used for objects:

fillers retrieves the role fillers at a role;
flexget(principia,fillers(has_author,initial),Result).
```
% binds Result to [russell,whitehead]
```

atleast retrieves the (qualifying) minimum restriction at a role;
flexget(group_publication,atleast(has_author),Result).
```
% binds Result to 3
```

atmost retrieves the (qualifying) maximum restriction at a role;
flexget(individual_publication,atmost(has_author),Result).
```
% binds Result to 1
```

tvf_filler retrieves the filler of a term-valued feature;
flexget(flex,tvf_filler(has_topic,initial),Result).
```
% binds Result to description_logics
```

all retrieves the value restriction at a role.
flexget(principia,all(has_author),Result).
```
% binds Result to famous
```

Finally we can retrieve the (most specific) concepts of which an object is an instance by using the methods CONCEPTS and MSC, as well as the objects which are instances of a concept by using INSTANCES:

concepts retrieves all concepts of which an object is an instance.

```
flexget(principia,concepts(initial),Result).
% binds Result to [ctop,publication,book]
```

msc retrieves the most specific concepts of which an object is an instance.

```
flexget(principia,msc(initial),Result).
% binds Result to [book]
```

instances retrieves all objects which are instances of a concept.

```
flexget(article,instances(initial),Result).
% binds Result to
[chinese_room,multi_pub,tractable_dl,obj_2]
```

Filter

The filter construct supports a limited form of representing second-order information by specifying properties of concepts and roles. Note that these properties are properties of concepts or roles and *not* of their instances, i.e. they are not inherited.

Filters are useful for several reasons: for one thing, they can be used to partition the conceptual hierarchy, e.g. by distinguishing different layers in the hierarchy, or by annotating which developer introduced which concepts, or by annotating which user will work with which concepts. The second major use of filters concerns the control of the FLEX inference behavior and will be explained in more detail below.

Filters have to be specified when terms are defined, as illustrated in the following example, in which we distinguish different between “geographical” subconcepts of conferences (european conferences vs international conferences) and topical subconcepts of conferences (AI conferences vs linguistic conferences):

```
   linguistic_conference  :<  conference with filter=topic_conf_type.
international_conference  :<  conference with filter=geo_conf_type.
     european_conference  :<  conference with filter=geo_conf_type.
                    ecai  :<  ai_conference and european_conference.
                   ijcai  :<  ai_conference and international_conference.
                  ecai96  ::  ecai.
                 ijcai95  ::  ijcai.
```

The effect of these filters is illustrated below:

```
% binds Result to [ai_conference,linguistic_conference,international_co
```

```
                                                  % binds Result to [
```

Reading and Dumping

So far we have typed in definitions, rules, and object descriptions interactively. When building a realistic model, however, the respective tells should rather be read from a file. If the FLEX system has been installed as described in Section A and started in the directory ‘flex’ you can read in an example file as follows:[6]

flexread(’DOCU/examples.model’).

Note that the file ‘examples.model’ starts with the command FLEXINIT, which initializes the knowledge base (i.e. all tells typed in previously are retracted) and sets the verbosity to INFO (see below). The system therefore outputs information about each tell it has read in.

Note further that FLEX performs a syntax check before processing the individual tells. If FLEX detects syntactic errors it produces a listing and asks the user whether she wants to read in the model anyway.

Though each individual tell is processed quickly by the system, reading in large models is usually rather time consuming. To avoid reading in a model each time you want to work with it, FLEX allows the dumping and loading of knowledge bases:

flexdump(’DOCU/examples.dump’).
flexload(’DOCU/examples.dump’).

It should be noted that FLEXDUMP will overwrite any existing file with the specified name.

When developing a large model it is in general useful to dump the stable part of the model and to use a combination of FLEXLOAD and FLEXREAD to test out the new/modified parts. Needless to say that in this case, the files to be read after loading the stable part should not contain the FLEXINIT command.

[6]The example file contains all the tells used in this manual up to this point, i.e. executing the commands in this subsection will give you the same knowledge base as if you had typed in all the tells individually.

State	*Settings*	*Setting after* FLEXINIT
class_objects	on, off	on
defspace_dbox	best,all	best
eval_dbox	local,global	local
introduction	forward,noforward	noforward
syntax_check	on,off	on
verbosity	silent,error,warning,info,trace	info

Figure 3.2: States, possible settings, and settign after FLEXINIT.

States

The behavior of the FLEX system can be controlled by setting various flags or states as summarized in the table in Figure 3.2. The state VERBOSITY, for example, controls the amount of information output by the system. The respective settings are obtained by typing the following commands:

```
flexstate(verbosity=silent).
flexstate(verbosity=error).
flexstate(verbosity=warning).
flexstate(verbosity=info).
flexstate(verbosity=trace).
```

FLEX performs auto-completion on states and settings, i.e. you can also type 'flexstate(v=i).' to set the verbosity on INFO.

To see the effect of the respective settings, just try the following object tells with different settings:

```
X :: scientific_publication and fiction. % this query fails
X :: at_conference:ijcai.
```

Note that the command FLEXINIT assigns each flag its default value, e.g. it sets the verbosity on INFO. If you prefer other settings, you thus have to specify these settings after the FLEXINIT if you read in the model from a file, or you can define your own init as a Prolog predicate, e.g.:

```
my_init :–
        flexinit,
        flexstate(v=s).
```

The state INTRODUCTION can be used to turn on/off the automatic introduction of undefined terms:

flexstate(introduction=forward).
flexstate(introduction=noforward).

Note that the forward introduction introduces undefined terms as primitive terms and that it is *not* possible to redefine these terms. Forward introduction is thus useful to quickly test the system behavior for small examples. For real models we strongly recommend to turn off the automatic introduction since otherwise typos are not detected. Therefore, automatic introduction is turned off by FLEXINIT.

The states CLASS_OBJECTS, DEFSPACE_DBOX and EVAL_DBOX are used to control the inference behavior of FLEX. The latter two concern weighted defaults and will be explained in Section 3.3. If CLASS_OBJECTS is set to state ON, objects will be classified wrt the conceptual hierarchy, i.e. their most specific subsuming concepts are computed. This is reasonable in information-system applications, in which retrieving instances is a frequent operation. If CLASS_OBJECTS is set to state OFF objects are not classified wrt the conceptual hierarchy which reduces the time needed for processing object tells. Note that neither the method INSTANCES nor a variable on the left-hand side of ?:/2 can be used ifCLASS_OBJECTS is set to OFF.

Controling Inferences

We end this section by briefly sketching how the inference behavior of FLEX can be controlled.[7] There are basically two ways of influencing the inference behavior:

1. It is possible to switch off inference rules completely, or to select between an application during normalization (forward reasoning) and during subsumption checking (backward reasoning).

2. When defining a role, the propagation behavior of the role can be specified by using the filter construct.

Note that both ways of controling the inference behavior presuppose a basic familiarity with the inference mechanisms underlying DL systems and are not yet customized towards end users (i.e. if you are a novice to DL you may safely skip this subsection).

You can change the application mode of an inference rule when using the graphical user interface (see Chapter 5. Click on the 'configuration' menu, then choose CHANGE RULES IN CLASS, then select a class (e.g. Objall) and modify the rule application.

In order to specify the propagation behavior of roles you have to use the built-in filter 'no_prop' as illustrated by the following examples:

[7] For the proof-theoretical foundation see Section C.3.

```
r1  :<  rtop with filter=[no_prop([1])].
c1  :<  ctop.
o1  ::  r1:o2 and all(r1,c1).
o2  ?:  c1.        % this query fails
r2  :<  rtop with filter=[no_prop([2])].
o1  ::  r2:o2.
o2  ?:  inv(r2):o1. % this query fails
```

The main idea of the 'no_prop' filter is to allow the user to switch of inferences whose results she knows she will never use. To switch of all propagations for a role, simply use the filter 'no_prop' with no argument, to switch of several propagations rules, put all of them in the list-argument of 'no_prop'.

3.3 Weighted Defaults

In this section we will briefly sketch the use of weighted defaults in a DL-model. It should be noted, however, that our experiences with weighted defaults in applications are still rather limited and that we are still investigating how to facilitate their use.

The basic idea of defaults is to represent rules which allow for exceptions. Their syntactic format is thus similar to strict rules, but whereas

$$c_1 \quad => \quad c_2$$

means that *every* object which is a 'c_1' also is a 'c_2', the corresponding default

$$c_1 \quad \sim N \sim > \quad c_2$$

allows for exceptions. This is usually paraphrased as: an object which is an instance of 'c_1' is also an instance of 'c_2' unless this is inconsistent with other information. Most of the research in Nonmonotonic Reasoning has focused on the question how this intuitive characterization can be turned into a formal semantics. The implementation of weighted defaults in FLEX is based on a preferential modeltheoretic semantics developed in [Quantz, Suska 94, Quantz 95] (see Appendix C.2).

In the following we will use defaults to model criteria for deciding whether to buy a book or not. We model the result of this decision with the feature

```
boolean  :=  oneof([yes,no]).
 buy_it  :<  domain(book) and range(boolean) and feature.
```

We can then specify defaults whose conclusions say wether to buy a book or not and whose premises are the criteria relevant for the decision.

One important criterion might be the price of a book, which can be modeled by

```
                       has_price    :<     domain(book) and range(number) and
                                           feature.
book and the(has_price,lt(10))    ~20~>   buy_it:yes.
```

The effect of this default can be illustrated by considering the following object tell:

```
book_1  ::  has_price:9.
```

Thus ‘book_1’ is an instance of the left-hand side of the above default. To trigger the application of the default we have to specify the defaults-option in the flexget-query:

```
flexget(book_1,fillers(buy_it,initial),box=defaults,Result).
                       % binds Result to [yes]
```

Note that the default can be straightforwardly overridden, e.g. adding

```
book_1  ::  buy_it:no.
```

yields

```
flexget(book_1,fillers(buy_it,initial),box=defaults,Result).
                        % binds Result to [no]
```

Besides this explicit overriding in descriptions, defaults can also be overridden by strict rules:

```
                                 written_in   :<   domain(publication) and
                                                   range(language) and feature.
book and the(written_in,unknown_language)    =>   buy_it:no.
```

Thus having

```
book_2  :: has_price:9.
```

yields

```
flexget(book_2,fillers(buy_it,initial),box=defaults,Result).
                       % binds Result to [yes]
```

but adding

```
book_2  ::  written_in:russian.
```

yields

```
flexget(book_2,fillers(buy_it,initial),box=defaults,Result).
% binds Result to [yes] flexget(book_2,fillers(buy_it,s1),box=defaults,Result).
                                                     % binds Result to [no]
```

since ‘russian’ is an unknown language in ‘s1’.

Finally, defaults cannot only be overridden by strict information but also by other defaults. Thus given only the above default we would buy any book costing less than 10$, regardless of author or topic. It seems more reasonable, however, to buy a book only if there are important reasons to do so, i.e. to set ‘buy_it’ to ‘no’ per default:

```
  book   ~100~>  buy_it:no.
book_3     ::    book and has_price:9.
```

Note that two defaults are applicable at ‘book_3’, one having weight 20, the other weight 100. Roughly speaking, the semantics for weighted defaults says that in case of conflicting defaults, default application has to be performed such that the sum of the weights of the overridden defaults is minimized.

```
flexget(book_3,fillers(buy_it,initial),box=defaults,Result).
                        % binds Result to [no]
```

Given the above model, we thus have to weight the defaults modeling criteria for buying a book such that their cumulative weight is greater than 100.

```
                  interesting_topic     :<     topic.
                     favorite_topic     :<     topic.
        natural_language_processing     :<     interesting_topic.
                philosophy_of_mind      :<     favorite_topic.
book and has_topic:interesting_topic   ~75~>   buy_it:yes.
   book and has_topic:favorite_topic  ~110~>   buy_it:yes.
       book and the(has_price,ge(50))  ~20~>   buy_it:no.
   book and some(has_author,famous)    ~20~>   buy_it:yes.
```

The effect of these defaults is summarized in Figure 3.3, some examples are given below:

has_topic	*has_price*	*has_author*	*Sum of “yes”-weights*	*Sum of “no”-weights*	*buy_it*
—	< 10	famous	40	100	no
—	< 10	—	20	100	no
—	$\geq 10, < 50$	famous	20	100	no
—	$\geq 10, < 50$	—	0	100	no
—	≥ 50	famous	20	120	no
—	≥ 50	—	0	120	no
interesting_topic	< 10	famous	115	100	yes
interesting_topic	< 10	—	95	100	no
interesting_topic	$\geq 10, < 50$	famous	95	100	no
interesting_topic	$\geq 10, < 50$	—	75	100	no
interesting_topic	≥ 50	famous	95	120	no
interesting_topic	≥ 50	—	75	120	no
favorite_topic	< 10	famous	150	100	yes
favorite_topic	< 10	—	130	100	yes
favorite_topic	$\geq 10, < 50$	famous	130	100	yes
favorite_topic	$\geq 10, < 50$	—	110	100	yes
favorite_topic	≥ 50	famous	130	120	yes
favorite_topic	≥ 50	—	110	120	no

Figure 3.3: Effects of defaults.

winograd	::	famous.
wittgenstein	::	famous.
book_4	::	has_topic:natural_language_processing and has_price:9 and has_author:winograd.
book_5	::	has_topic:natural_language_processing and has_price:20.
book_6	::	has_topic:natural_language_processing and has_price:20 and has_author:winograd.
book_7	::	has_topic:philosophy_of_mind and has_price:20.
book_8	::	has_topic:philosophy_of_mind and has_price:50 and has_author:wittgenstein.
book_9	::	has_topic:philosophy_of_mind and has_price:50.

These tells yield the following results:[8]

[8]We are still working on an adequate interface for default reasoning. The predicate dbox_print_results(Object,Situation) outputs some information about applied and overridden defaults.

```
flexget(book_4,fillers(buy_it,initial),box=defaults,Result).
                                     % binds Result to [yes]
dbox:dbox_print_results(book_4,initial).
flexget(book_5,fillers(buy_it,initial),box=defaults,Result).
                                      % binds Result to [no]
dbox:dbox_print_results(book_5,initial).
flexget(book_6,fillers(buy_it,initial),box=defaults,Result).
                                      % binds Result to [no]
dbox:dbox_print_results(book_6,initial).
flexget(book_7,fillers(buy_it,initial),box=defaults,Result).
                                     % binds Result to [yes]
dbox:dbox_print_results(book_7,initial).
flexget(book_8,fillers(buy_it,initial),box=defaults,Result).
                                     % binds Result to [yes]
dbox:dbox_print_results(book_8,initial).
flexget(book_9,fillers(buy_it,initial),box=defaults,Result).
% binds Result to [no] dbox:dbox_print_results(book_9,initial).
```

We are still working on an adequate interface for default application. The following example

```
dbox:dbox_print_results(book_8,initial).
```

Finally, we will briefly describe two states which can be used to control the inference behavior of the DBox. The state DEFSPACE_DBOX can be set to BEST or ALL. If it is set to BEST only the best default space is computed, if it is set to ALL all default spaces are computed. The latter is useful for debugging purposes.

The state EVAL_DBOX can be set to LOCAL or GLOBAL. If it is set to LOCAL defaults will be optimized locally at each object only. Note that this is not correct from a semantic point of view but is sufficient for most applications.

Chapter 4

Inference Algorithms

In this chapter we sketch the main inference algorithms implemented in FLEX. We begin by describing the algorithms used for the strict part of FLEX, namely normalization, subsumption checking, and object level reasoning. We then present the main structure of the algorithm used for computing default spaces.

4.1 Computing Subsumption

Inferences in FLEX

The main inferential task of the FLEX system is to answer queries of the form

$$\begin{array}{ccl} \mathrm{o} & ?: & \mathrm{c\ in\ s} \\ \mathrm{t}_1 & ?< & \mathrm{t}_2 \end{array}$$

wrt a modeling containing formulae of the form

$$\begin{array}{ccl} \mathrm{t}_n & := & \mathrm{t} \\ \mathrm{c}_1 & => & \mathrm{c}_2 \\ \mathrm{s}_1 & \leq & \mathrm{s}_2 \\ \mathrm{o} & :: & \mathrm{c\ in\ s} \end{array}$$

Two things are important to note:

1. The FLEX system already performs inferences when reading in a modeling. There are two major inference components, namely the *classifier* and the *recognizer*. The classifier checks subsumption between the terms defined in the terminology and thus computes the *subsumption hierarchy*. The recognizer determines for each object which concepts it instantiates and thus computes the *instantiation index*.

2. For answering both kinds of queries, the same method can be used, namely *subsumption checking*. Thus when answering a query o ? : c ins, the system checks whether the normal form derived for o in s is subsumed by c.

Though the recognizer thus uses the classifier to perform its task, there is an important difference between the two components. Whereas the classifier performs only "local" operations, the recognizer has to perform "global" operations. This distinction can be illustrated by briefly skeching the algorithmic structure of both components.

Given a list of definitions, the classifier takes each definition and compares it with all previously processed definitions, thereby constructing a directed acyclic graph called the subsumption hierarchy. Thus the concept classifier is a function 'Concept $\times$ DAG $\rightarrow$ DAG', where the initial DAG contains the nodes **ctop** and **cbot**. Locality here means that classifying a concept has no impact on previous classification results, i.e. classifying concept c_3 has no impact on the subsumption relation between c_1 and c_2.

Recognition, on the other hand, has global effects. Basically, the recognizer processes a list of descriptions and computes for each object which concepts it instantiates, i.e. it is a function 'Description $\times$ Index $\times$ DAG $\rightarrow$ Index'. Nonlocality here means that recognition triggered by a description 'o_1 :: c in s' can lead to changes in the instantiation index for some other object o_2, as exemplified by

$$\begin{array}{lll} o_1 & :: & r{:}o_2 \text{ in s} \\ o_1 & :: & \forall r{:}c \text{ in s} \end{array}$$

Here processing the second description includes the derivation of o_2 :: c in s.

Note that another distinction between classification and recognition is thus that there is exactly one definition for each term in the terminology, whereas objects can be described incrementally, i.e. we can have several descriptions for an object in a situation.

In this report we will describe the algorithmic structure of *normalization*, *subsumption checking*, and *object-level propagation*. Before doing so in the following sections, we will briefly present the proof-theoretical basis of these algorithms.

Tableaux-Based Algorithms vs Normalize-Compare Algorithms

The first classifiers for DL were specified as *structural subsumption* algorithms [Schmolze, Israel 83]. The basic idea underlying structural subsumption is to transform terms into canonical normal forms, which are then structurally compared. Structural subsumption algorithms are therefore also referred to as *normalize-compare* algorithms. Note that there is a general tradeoff between normalization and comparison: the more inferences are drawn in normalization, the less inferences have to be drawn in comparison, and vice versa.

$$\begin{array}{ccccccc}
\Gamma & \models_{DL} & \gamma & \text{iff} & \overline{\Gamma} & \models_{FOL} & \overline{\gamma} \\
 & \text{iff} & & & & \text{iff} & \\
\Gamma & \vdash_{DL} & \gamma & \text{iff} & \overline{\Gamma} & \vdash_{SC} & \overline{\gamma}
\end{array}$$

Figure 4.1: The commutative diagram underlying the sequent-style approach to inference rule derivation.

There is one severe drawback of normalize-compare algorithms—though it is in general straightforward to prove the correctness of such algorithms there is no method for proving their completeness. In fact, most normalize-compare algorithms are incomplete which is usually demonstrated by giving examples for subsumption relations which are not detected by the algorithm [Nebel 90].

At the end of the 1980's *tableaux methods*, as known from FOL (cf. [Sundholm 83, p. 180ff]), were applied to DL (e.g. [Donini et al. 91a, Donini et al. 91b]). The resulting subsumption algorithms had the advantage of providing an excellent basis for theoretical investigations. Not only was their correctness and completeness easy to prove, they also allowed a systematic study of the decidability and the tractability of different DL dialects.

The main disadvantage of tableaux-based subsumption algorithms is that they are not constructive but rather employ refutation techniques. Thus in order to prove the subsumption $\mathrm{c}_2 \sqsubseteq \mathrm{c}_1$ it is proven that the term $\mathrm{c}_1 \sqcap \neg\, \mathrm{c}_2$ is inconsistent, i.e. that $\mathrm{o} :: \mathrm{c}_1 \sqcap \neg\, \mathrm{c}_2$ is not satisfiable. In most existing systems, on the other hand, inference rules are more seen as production rules, which are used to pre-compute part of the consequences of the initial information. This corresponds more closely to Natural Deduction or Sequent Calculi, two deduction systems also developed in the context of FOL.

A third alternative, combining advantages of the normalize-compare approach and tableaux-based methods has been proposed in [Royer, Quantz 92]. The basic idea is to use *Sequent Calculi* instead of tableaux-based methods for the characterization of the deduction rules. Like tableaux methods, sequent calculi provide a sound logical framework, but whereas tableaux-based methods are refutation based, i.e. suitable for theorem checking, sequent calculi are constructive, i.e. suitable for theorem proving.

The methodology for constructing a sequent-style proof theory for DL is straightforward and summarized by the commutative diagram shown in Figure 4.1. Since DL are subsets of First-Order Logic (FOL), we can translate a DL formula γ into an FOL formula ($\overline{\gamma}$). That is we know, due to the completeness of the Sequent Calculus ($\vdash_{SC}$) for FOL:

$$\Gamma \models_{DL} \gamma \quad \text{iff} \quad \overline{\Gamma} \models_{FOL} \overline{\gamma}$$

$$\overline{\Gamma} \models_{FOL} \overline{\gamma} \quad \text{iff} \quad \overline{\Gamma} \vdash_{SC} \overline{\gamma}$$

If we now can show that

$$\overline{\Gamma} \vdash_{SC} \overline{\gamma} \quad \text{iff} \quad \Gamma \vdash_{DL} \gamma$$

we immediately get the desired completeness

$$\Gamma \models_{DL} \gamma \quad \text{iff} \quad \Gamma \vdash_{DL} \gamma$$

Thus the main task is to prove that for every SC proof of an FOL translation of a DL formula γ there is a corresponding DL proof of γ.

Applying this methodology one obtains inference rules like

$$\begin{array}{rcl} \top \sqsubseteq \text{c} & \rightleftharpoons & \top \sqsubseteq \forall\text{r:c} \\ \text{c}_2 \sqcap \text{c}_1 \sqsubseteq \bot \text{ , r}_1 \sqsubseteq \text{r}_2 & \rightharpoonup & \forall\text{r}_2\text{:c}_2 \sqcap \geq p \text{ r}_1\text{:c}_1 \sqsubseteq \bot \end{array}$$

Note that this format is sufficient for a theoretical characterization of a deduction system, i.e. given a set of inference schemata Σ we can define a least fixed point Φ^{ω} by taking the logical closure of a set of formulae Γ under Σ. We can then say that $\Gamma \vdash_{\Sigma} \gamma$ iff $\gamma \in \Phi^{\omega}(\Gamma)$.

Though we can study formal properties like soundness or completeness, i.e. the relation between $\Gamma \vdash_{\Sigma} \gamma$ and $\Gamma \models \gamma$, on the basis of this characterization, we need an additional control strategy for turning the deduction system into an algorithm. The main reason for this is that $\Phi^{\omega}(\Gamma)$ is not finite. The sequent-style approach thus falls into two separate phases:

1. Derivation of a complete axiomatization by systematically rewriting FOL proofs.

2. Specification of a control strategy to turn the complete axiomatization into an algorithm.

In the following we will concentrate on the second task, relying on the axiomatization derived in [Royer, Quantz 93]. Note that this axiomatization is not complete for the DL underlying FLEX. As shown in [Royer, Quantz 93], the complexity of deriving a complete axiomatization differs considerably wrt the investigated DL fragments:

1. A complete axiomatization of subsumption is specified for a DLcontaining only concept-forming operators without equality and no role-forming operators.

2. A complete axiomatization is also given for role subsumption.

Concept Atom	*Abstract Syntax*	*Role Atom*	*Abstract Syntax*
ctop	$\top$	rtop	$\top^{r}$
cbot	$\bot$	rbot	$\bot^{r}$
prim(C)	c_p	prim(R)	r_p
atleast(N,R,C)	$\geq n$ r:c	domain(C)	c|r
atmost(N,R,C)	$\leq n$ r:c	range(C)	r|c
all(R,C)	$\forall$r:c	comp([R1,R2])	$r_1.r_2$
rvm_eq([R1,R2])	$r_1=r_2$	inv(R)	r^-
R:[O])	r:o		
oneof([O1,...,On])	$\{o_1, \ldots, o_n\}^\vee$		

Figure 4.2: Correspondence between atoms and abstract syntax.

3. The complexitiy of deriving complete axiomatizations is considerably higher, however, for
   (a) subsumption in DL containing role-forming operators and/or concept-forming operators involving equality
   (b) object-level reasoning

Though this is in fact a rather negative result, it does not make the sequent-style approach useless. Based on the sequent-style rules, *flexible inference systems* can be implemented, which allow to control the inference strategy by specifying whether a rule should be applied during normalization, during comparison, or not at all. The sequent-style rules are thus used to formally characterize incomplete reasoning (see also [Royer, Quantz 94]).

Normalization

As should have become obvious from the general description above, normal forms play a crucial role in the implementation of DL inference algorithms. The FLEX system computes for each concept name, role name, and object a normal form, which is the basis for further processing. We will now briefly sketch

1. the format of normal forms;
2. the transformation of DL terms into normal forms;
3. the structure of normalization rules.

Let us start with the *format* of normal forms. The basic building blocks of normal forms are so-called *atoms*, which correspond to the term-building operators of the

$a_1 \wedge a_2$	nf([a_1, a_2],[ctop])
$a_1 \vee a_2$	nf([ctop],[nf([a_1],[ctop]),nf([a_2],[ctop])])
$a_1 \wedge (a_2 \vee a_3)$	nf([a_1],[nf([a_2],[ctop]),nf([a_3],[ctop])])

Figure 4.3: Examples for the normal forms based on [Kasper 87].

Concept Atom	*Negation*	*Role Atom*	*Negation*
ctop	cbot	rtop	rbot
prim(C)	neg_prim(C)	prim(R)	neg_prim(R)
atleast(N,R,C)	atmost(N-1,R,C)	domain(C)	domain(NEG(C))
all(R,C)	atleast(1,R,NEG(C))	range(C)	range(NEG(C))
rvm_eq([R1,R2])	neg_rvm_eq([R1,R2])	comp([R1,R2])	neg_comp([R1,R2])
fillers(R,[O])	neg_fillers(R,[O])	inv(R)	neg_inv(R)
oneof([O1,...,On])	neg_oneof([O1,...,On])		

Figure 4.4: Negation of atoms.

DL. Figure 4.2 shows the correspondence between atoms and the abstract syntax used in Chapter C. Note that the R's and C's occurring in the atoms are themeselves normal forms of roles and concepts. One way of formally defining atoms and normal forms is thus by means of a parallel inductive definition, as done in [Royer, Quantz 92].

For languages not containing negation or disjunction, a normal form is simply a set of atoms. Since FLEX contains both disjunction and negation, however, we need a more complicated format, however. Kasper has proposed a format of disjunctive normal forms, which allows fast detection of unification failures [Kasper 87]. The basic idea is to separate between the non-disjunctive part of a normal form and the disjunctive part. The non-disjunctive part of a normal form is simply a list of atoms, the disjunctive part is a list of normal forms. Since 'ctop', 'cbot', 'rtop', and 'rbot' are considered full-fledged normal forms, this recursive definition is well founded.

Note that each term can easily be transformed into a normal form by "eliminating" conjunctions, disjunctions, and negations. The examples in Figure 4.3, in which a_i stand for arbitrary atoms, illustrate this transformation of terms into normal forms. Note further that negation can be completely eliminated since the set of atoms is closed under negation. Figure 4.4 shows for each atom its negation. Thus given a concept definition 'c$_n$:= c' or an object description 'o :: c in s', the system transforms the term 'c' into a normal form. This normal form is then further processed by applying normalization rules. The basic idea of normalization is to make information implicitly contained in a normal form explicit, i.e.

NORMALIZE: NF $\rightarrow$ NF

In principle, we can distinguish 3 types of normalization rules as illustrated by the following examples:

1. Some inference rules are already used in the transformation of external terms into internal normalform format, e.g.

$$\rightharpoonup \quad \neg\forall r : c \doteq \exists r : \neg c$$

2. Most inference rules yield a subsumption formula where the subsuming term can be represented as an atom and the subsumed term as a list of atoms, e.g.

$$\begin{array}{rcl} c_1 \sqcap c_2 \sqsubseteq c_3, & & \\ r_2 \sqsubseteq r_1, r_2 \sqsubseteq r_3 & \rightharpoonup & \exists r_1 : c_1 \sqcap \forall r_2 : c_2 \sqsubseteq \exists r_3 : c_3 \end{array}$$

3. Some inference rules yield a subsumption formula containing disjunctions in the subsuming or subsumed term, e.g.

$$\rightharpoonup \quad \forall r.r^{-} : c \sqsubseteq c \sqcup \leq 0 r : \top$$

Thus the first type of normalization rules is “hard-wired” into the parser and cannot be controlled from the outside. This is necessary to guarantee a format of normalforms which makes further processing more efficient (cf. the restriction to Negation Normal Forms in certain variants of the Sequent Calculus).

For the second type of normalization rules, a straightforward normalization strategy is used. The conjunctive part of a normalform, i.e. a list of atoms, is processed one by one, using the currently processed atom as trigger and then looking for the other atoms needed in the triggering conditions. Formally, these normalization rules have the general format

$$\alpha_1, \ldots, \alpha_n \quad \rightharpoonup \quad \alpha$$

i.e. if the atoms $\alpha_1, \ldots, \alpha_n$ are contained in the conjunctive part of a normalform, then α is added to this conjunctive part.

Roughly speaking, a predicate ‘norm(TriggerAtom,CNF,AddAtom)’ is used to realize this type of normalization rules, i.e. we have predicates like

```
norm(some(R1,C1),Con,some(R1,C1 and C2)) :–
      active_norm_rule(23),
      member(all(R2,C2),Con),
      subsumes(R2,R1).
```

Note that the information whether this rule is applied in the normalization phase is provided by 'active_norm_rule'. Thus a configuration specifying a particular inference strategy for FLEX consists of a list of declarations for active normalization or subsumption rules.

The third type of normalization rules is more difficult to realize, since it involves either checking of disjunctions in the triggering conditions or adding of complex information to a normalform.

Subsumption Checking

The task of subsumption checking is to decide whether a normal form subsumes another normal form, i.e.

$$\text{SUBSUMES: NF} \times \text{NF} \rightarrow \text{BOOL}$$

As has been mentioned above, there is a general trade-off between normalization and subsumption checking. The more inferences are performed during normalization, the less inferences have to be drawn during subsumption checking. In principle it is possible to produce normal forms which guarantee that subsumption checking only has to test subsumption between individual atoms. Such *vivid* normal forms have been presented for a simple fragment of DL in [Royer, Quantz 92, Sect. 5].

The disadvantage of such vivid normal forms is that they require the application of many normalization rules, making information explicit which might never be needed. Performing inferences during subsumption checking on the other hand guarantees that inferences are only drawn when actually needed.[1]

Basically, subsumption checking between normal forms is ultimately reduced to subsumption checks between atoms, but includes also special subsumption rules for disjunctive parts, non-disjunctive parts, etc. Figure 4.5 shows the reduction of subsumption between normal forms to subsumption between parts of normal forms and ultimately atoms.[2]

Object-Level Reasoning

As already indicated above, object-level reasoning is inherently non-local and it is therefore useful to distinguish between a local phase and a non-local phase in object-level reasoning.

[1] In general, a mixed strategy is needed, which ensures both efficient performance and detection of inconsistency already during normalization.

[2] To keep the presentation simple, subsumption rules are only given for concepts but not for roles.

```
subsumes_nf_nf(ctop,_).
subsumes_nf_nf(_,cbot).
subsumes_nf_nf(nf(Con,Dis),NF) :–
        subsumes_con_nf(Con,NF), subsumes_dis_nf(Dis,NF).

subsumes_con_nf([Atom],NF) :–
        subsumes_atom_nf(Atom,NF).
subsumes_con_nf([Atom|Atoms],NF) :–
        subsumes_atom_nf(Atom,NF), subsumes_con_nf(Atoms,NF).

subsumes_atom_nf(Atom,nf(Con,Dis)) :–
        subsumes_atom_con(Atom,Con); subsumes_atom_dis(Atom,Dis).

subsumes_atom_con(Atom1,Con) :–
        member(Atom2,Con), subsumes_atom_atom(Atom1,Atom2).

subsumes_atom_dis(Atom,[NF]) :–
        subsumes_atom_nf(Atom,NF).
subsumes_atom_dis(Atom,[NF|NFs]) :–
        subsumes_atom_nf(Atom,NF), subsumes_atom_dis(Atom,NFs).

subsumes_dis_nf(Dis1,nf(Con,Dis2)) :–
        subsumes_dis_con(Dis1,Con); subsumes_dis_dis(Dis1,Dis2).

subsumes_dis_con(Dis,Con) :–
        member(NF,Dis), subsumes_nf_con(NF,Con).

subsumes_nf_con(nf(Con1,Dis),Con2) :–
        subsumes_con_con(Con1,Con2), subsumes_dis_con(Dis,Con).

subsumes_con_con([Atom],Con) :–
        subsumes_atom_con(Atom,Con).
subsumes_con_con([Atom|Atoms],Con) :–
        subsumes_atom_con(Atom,Con), subsumes_con_con(Atoms,Con).

subsumes_dis_dis([NF],Dis) :–
        subsumes_nf_dis(Nf,Dis).
subsumes_dis_dis([NF|NFs],Dis) :–
        subsumes_nf_dis(Nf,Dis), subsumes_dis_dis(NFs,Dis).
```

Figure 4.5: Reduction of subsumption.

In the local phase we determine for an object the *most specific concept* it instantiates. This can be done by using the standard normalize and compare predicates. Thus we normalize the description of an object thereby obtaining a normal form and compare it with the normal forms of the concepts in the hierarchy. In addition to this standard classification we also have to apply rules when processing objects. This is achieved by applying all rules whose left-hand sides subsume the object's normal form. After this application the normal form is again normalized and classified until no new rules are applicable [Owsnicki-Klewe 88].

In the non-local phase we have to propagate information to other objects. There are six basic rules for propagation:

$$\begin{aligned}
\text{o}_1 :: \forall \text{r:c in s}, \text{o}_1 :: \text{r:o}_2 \text{ in s} &\rightharpoonup \text{o}_2 :: \text{c in s} \\
\text{o}_1 :: \forall \text{r:}\{\text{o}_2, \ldots, \text{o}_n\}^{\vee} \text{ in s}, \text{o}_2 :: \text{c in s}, \ldots, \text{o}_n :: \text{c in s} &\rightharpoonup \text{o}_1 :: \forall \text{r:c in s} \\
\text{o}_1 :: \text{r:o}_2 \text{ in s}, \text{o}_2 :: \text{c in s} &\rightharpoonup \text{o}_1 :: \text{r}|_{\text{C}}\text{:o}_2 \text{ in s} \\
\text{o}_1 :: \text{r}_1\text{:o}_2 \text{ in s}, \text{o}_2 :: \text{r}_2\text{:o}_3 \text{ in s} &\rightharpoonup \text{o}_1 :: \text{r}_1.\text{r}_2\text{:o}_3 \text{ in s} \\
\text{o}_1 :: \text{f:o}_2 \sqcap \text{f.r:o}_3 \text{ in s} &\rightharpoonup \text{o}_2 :: \text{r:o}_3 \text{ in s} \\
\text{o}_1 :: \text{r:o}_2 \text{ in s} &\rightharpoonup \text{o}_2 :: \text{r}^{-}\text{:o}_1 \text{ in s}
\end{aligned}$$

We will briefly sketch three important aspects of propagation. First, consider the first rule, called *forward propagation*, and the question of how to trigger application of this rule. The easiest way of realizing forward propagation is to test whether an object normal form contains atoms 'all(R,C)' and 'fillers(R,O)' and then to propagat 'C' to all members of 'O'.

The problem of this naive approach is that objects can be described incrementally and can be reclassified in the course of porpagation. It would be rather inefficient to propagate the same value restriction to the same filler over and over again, whenever some other part of the normal form of the object changes. To avoid superfluous propagations, only new information at an object can trigger propagations.

Second, consider the second propagation rule, called *backward propagation*. For one thing it is advantageous to apply this rule during subsumption checking rather then during normalization. This is due to the fact 'c' is underdetermined in the propagation rule, i.e. we could try arbitrary concepts and abstract value restrictions which will never be used.

Now assume we apply the rule during subsumption checking. This means that when classifying an object and checking whether it is subsumed by 'all(R,C)' we have to check whether all fillers for 'R' are known (i.e. whether 'R' is "closed"), and if so whether all fillers are instances of 'C'.

Things are further complicated due to incremental descriptions. Assume three tells in the following order:

$$\begin{array}{ccl} \forall r{:}c_1 & => & c_2 \\ o_1 & :: & \leq 1\ r \sqcap r{:}o_2 \text{ in s} \\ o_2 & :: & c_1 \text{ in s} \end{array}$$

When classifying o_1, we cannot prove that it is an instance of $\forall r{:}c_1$ since o_2 is not yet an instance of c_1. Hence the rule is not applied. The later description 'o_2 :: c_1 in s', however, makes o_1 an instance of $\forall r{:}c_1$ and the rule applicable. Again, a naive implementation in which a change at an object o_1 leads to a reclassification of all objects at which o_1 is a role filler is highly inefficient. Instead, the reclassification of o_1 is triggered by an "exploding type bomb".[3]

When classifying o_1 we record that o_1 could not be recognized as an instance of $\forall r{:}c_1$ since o_2 could not be recognized as an instance of c_1. Thus as soon as o_2 becomes an instance of c_1, the type bomb explodes and o_1 is reclassified.

Finally, consider the rule for composition. Again, applying it automatically during normalization leads to the derivation of irrelevant information. The FLEX system offers the possibility to turn of propagation rules for specific roles.

These examples where meant to illustrate the complexity of object-level reasoning and some of the mechanisms offered by FLEX to reduce its complexity. In spite of these mechansims object-level reasoning is still rather time-consuming. algorithm for interpretation presented in The most promising solution to this efficiency problem consists in a parallelization of propagation as described in [Bergmann, Quantz 95].

4.2 Computing Default Spaces

In this section we briefly describe an algorithm implementing the proof theory for weighted defaults developed in Chapter C. The basic functionality of this algorithm is to compute *default situations* with minimal score from a given situation. This is realized by succesively extending the initial situation, i.e. by recursively

1. determining applicable defaults and
2. applying as many of the applicable defaults as possible

until no defaults are applicable anymore.[4] The result of this recursive application of defaults is thus a situation representing a default space. In a second step alternative situations (default spaces) are constructed. The algorithm thus performs a *depth-first* strategy.

The basic data structure used for representing default spaces has the form

```
def_space(ID,Sit,In,Out,Missing,Prev,Score)
```

[3]This terminology goes back to Bob MacGregor (personal communication).

[4]Note that this basic process is similar to the application of strict rules [Owsnicki-Klewe 88].

where ‘ID’ is the identifier of the default space and ‘Prev’ is the identifier of the default space from which it was constructed (this information is used for backtracking). ‘Sit’ is the identifier of the default situation containing the information represented by the default space. ‘In’, ‘Out’, and ‘Missing’ are (ordered) sets of atoms, where an atom has the form ‘od(Object,Default,Weight)’. Their respectiv meaning is

$$\begin{array}{rcl}
\text{od(o,}\delta\text{,w(}\delta\text{))} \in \text{In} & \Rightarrow & \Gamma \mathrel{|\!\approx^k} \text{o} :: \delta_p \sqcap \delta_c \text{ in s} \\
\text{od(o,}\delta\text{,w(}\delta\text{))} \in \text{Out} & \Rightarrow & \Gamma \mathrel{|\!\approx^k} \text{o} :: \delta_p \text{ in s} \wedge \\
 & & \Gamma \mathrel{|\!\not\approx^k} \text{o} :: \delta_c \text{ in s} \\
\text{od(o,}\delta\text{,w(}\delta\text{))} \in \text{Missing} & \text{iff} & \Gamma \mathrel{|\!\approx^k} \text{o} :: \delta_p \text{ in s} \wedge \\
 & & \text{od(o,}\delta\text{,w(}\delta\text{))} \notin \text{In} \cup \text{Out}
\end{array}$$

Finally, ‘Score’ represents the negative score of the default space, i.e. we have

$$\text{Score} = \Sigma_{\text{od(o,}\delta\text{,}w_i\text{)} \in \text{Out}} w_i$$

The main algorithm for computing default situations is shown in Figures 4.6 and 4.7. I will now briefly describe the functionality of the main predicates and then illustrate the algorithm by giving an example.

The top-level predicate is ‘compute_default_spaces’, which takes a situation as input and returns a list of minimal default situations. It stores an initial default space, then computes a first complete default space, searches for alternative default spaces, and finally extracts the default situations from the default spaces.

The predicate ‘store_space’ gets most information about the default space passed as arguments. The only information it computes is the “missing” defaults. The predicate ‘missing_defaults’ is not shown in the figures, but its realization can be easily described. We first compute for each object the applicable defaults, e.g. by retrieving the most specific concepts of the object and computing all super concepts which are left-hand sides of defaults. The missing defaults then are those which are not yet in the ‘In’ or ‘Out’ lists of the default space.[5]

Completion of a default space is a recursive process. We successively extend the space until no defaults are missing anymore. Extension of a default space is achieved by adding as many of the missing defaults as possible, which is realized by the predicate ‘add_missing’. This predicate recursively processes the atoms in ‘Missing’ by calling ‘add_atom’. The predicate ‘add_atom(od(o,δ,w(δ)))’ tries to add the object description ‘o :: δ_c in s’. If this description can be consistently added it succeeds (and the respective information is part of ‘Sit’), if it cannot be consistently added it fails. Depending on the success of ‘add_atom’, the atom itself is included either in ‘In’ or in ‘Out’, and in the latter case the current score

[5]Note that this talk of missing defaults is abbreviatory. To be precise we would have to talk about atoms ‘od(o,δ,w(δ))’ since this is the format of the members of ‘In’, ‘Out’, and ‘Missing’.

```
compute_default_spaces(Sit,DefSits) :–
    store_space(Sit,[],[],nil,0,InitSpace),
    complete_space(InitSpace,_,0,FirstSpace,Score),
    alternatives(Space,Score,[Space],BestSpaces,_),
    extract_sits(BestSpaces,DefSits).

store_space(Sit,In,Out,Prev,Score,ID) :–
    all_sit_objects(Sit,Objects),
    append(In,Out,Checked),
    missing_defaults(Objects,Sit,Checked,[],Missing),
    new_space(ID),
    assert(def_space(ID,Sit,In,Out,Missing,Prev,Score)).

complete_space(Space,_,Current,Space,Current) :–
    ds_missing(Space,[]),!.
complete_space(Space,BestScore,CurrentScore,FinSpace,FinScore) :–
    extend_space(Space,BestScore,CurrentScore,ExtSpace,NewScore),
    complete_space(ExtSpace,BestScore,NewScore,FinSpace,FinScore).

extend_space(Space,BestScore,CS,ExtSpace,FScore) :–
    ds_sit_in_out_missing(Space,Sit,In,Out,Missing),
    sit_extend(Sit,NewSit),
    add_missing(Missing,NewSit,In,Out,Space,BestScore,CS,ExtSpace,FScore).

add_missing([],Sit,In,Out,Space,_,Score,ExtSpace,Score) :–
    store_space(Sit,In,Out,Space,Score,ExtSpace).
add_missing([Atom|Atoms],Sit,In,Out,Space,Best,CS,ESpace,FScore) :–
    (add_atom(Atom,Sit) ->
        NewIn = [Atom|In],
        NewOut = Out,
        NewScore = CS;
            NewOut = [Atom|Out],
            NewIn = In,
            Atom = od(_,_,Weight),
            NewScore is CS + Weight),
    score_okay(NewScore,Best),
    add_missing(Atoms,Sit,NewIn,NewOut,Space,Best,NewScore,ESpace,FScore).
```

Figure 4.6: First part of the default-space algorithm.

```
alternatives(Space,BestScore,BestSpaces,FinSpaces,FinScore) :–
        check_sisters(Space,BestScore,BestSpaces,NewSpaces,NewScore),
        ds_prev(Space,PrevSpace),
        (PrevSpace = nil ->
                FinSpaces = NewSpaces,
                FinScore = NewScore;
                        alternatives(PrevSpace,NewScore,NewSpaces,FinSpaces,FinScore)).

check_sisters(Space,BestScore,BestSpaces,FinSpaces,FinScore) :–
        in_out_pattern(Space,In,Out),
        next_sister(Space,In,Out,BestScore,Sister,NextScore),
        integrate_space(Sister,NextScore,BestSpaces,BestScore,NewSpaces,NewScore),
        alternatives(Sister,NewScore,NewSpaces,FinSpaces,FinScore),
        !.
check_sisters(_,BestScore,Spaces,Spaces,BestScore).

next_sister(Space,In,Out,BestScore,FinSpace,FinScore) :–
        next(In,Out,NextIn,NextOut),
        (build_sister(Space,BestScore,NextIn,NextOut,Sister,CurrentScore),
        complete_space(Sister,Score,CurrentScore,FinSpace,FinScore),!;
                next_sister(Space,NextIn,NextOut,BestScore,FinSpace,FinScore)).

build_sister(Space,BestScore,NextIn,NextOut,Sister,Score) :–
        ds_prev(Space,Prev),
        ds_sit_in_out_score(Prev,Sit,In,Out,PrevScore),
        add_score(NextOut,PrevScore,Min),
        score_okay(Min,BestScore),
        append(Out,NextOut,NewOut),
        sit_extend(Sit,NewSit),
        add_missing(NextIn,NewSit,In,NewOut,Prev,BestScore,Min,Sister,Score).
```

Figure 4.7: Second part of the default-space algorithm.

o	::	c$_1$ **and not**(c$_3$) **and** c$_4$ **and** c$_5$ in s
δ_1: c$_1$	$\leadsto_{10}$	c$_2$
δ_2: c$_2$	$\leadsto_{15}$	c$_3$
δ_3: c$_1$	$\leadsto_{15}$	**not**(c$_6$)
δ_4: c$_4$	$\leadsto_{10}$	c$_6$
δ_5: c$_5$	$\leadsto_{10}$	c$_6$
δ_6: c$_1$	$\leadsto_{20}$	c$_7$
δ_7: c$_7$	$\leadsto_{20}$	c$_8$
δ_8: c$_6$	$\leadsto_{1}$	**not**(c$_8$)

Figure 4.8: A sample modeling for illustrating the default-space algorithm.

is increased by ‘w(δ)’. Note that before processing the remaining atoms it is checked whether the current score does not already exceeds the best score obtained so far.

Having computed a first default space, alternatives are checked. In general, ‘alternatives’ receives as input the currently investigated default space, the best score obtained so far, as well as all default spaces constructed so far with this best score. It returns a (possibly new) set of optimal default spaces and the corresponding score. ‘alternatives’ proceeds in two steps: first the sister nodes of ‘Space’ are evaluated, then backtracking to the previous space is performed and alternatives of this previous space are computed (recursive call of ‘alternatives’). The recursion ends when the initial default space is reached (‘PrevSpace = nil’).

The sisters are again evaluated by a depth-first search. A sister is constructed and completed, then alternatives are checked. Note that ‘next_sister’ fails, if the respective spaces exceed the best score. The second clause of ‘check_sisters’ therefore guarantees successful termination.

Construction of a sister is achieved by ‘next_sister’. The basic idea is to take the current distribution of ‘Missing’ (in ‘PrevSpace’) into ‘In’ and ‘Out’ (computed by ‘in_out_pattern’) and to compute the next distribution (‘next’). This next distribution is passed to ‘build_sister’ which checks the minimal score of the new distribution and then tries to add as many elements of ‘NextIn’ as possible (by calling ‘add_missing’). The resulting sister is then completed by ‘complete_space’. If this does not produce a default space with sufficiently low score, ‘next_sister’ is called recursively and tries the next distribution.

For illustration consider the example modeling shown in Figure 4.8. In Figure 4.9 the default spaces generated by the algorithm are shown. To simplify the presentation the entries for ‘In’, ‘Out’, and ‘Missing’ only contain the defaults instead of the complex atoms ‘od(o,δ,w(δ))’. Note that the respective entries are ordered, i.e. defaults with higher weight precede defaults with lower weight. For

ID	IN	OUT	MISSING	PREV	SCORE
1	$\emptyset$	$\emptyset$	$\delta_6, \delta_3, \delta_5, \delta_4, \delta_1$	nil	0
2	$\delta_6, \delta_3, \delta_1$	δ_5, δ_4	δ_7, δ_2	1	20
3	$\delta_7, \delta_6, \delta_3, \delta_1$	$\delta_2, \delta_5, \delta_4$	$\emptyset$	2	35
4	δ_6, δ_3	$\delta_5, \delta_4, \delta_1$	δ_7	1	30
5	$\delta_7, \delta_6, \delta_3$	$\delta_5, \delta_4, \delta_1$	$\emptyset$	4	30
6	$\delta_6, \delta_5, \delta_4, \delta_1$	δ_3	$\delta_7, \delta_2, \delta_8$	1	15
7	$\delta_6, \delta_5, \delta_4$	δ_3, δ_1	δ_7, δ_8	1	25
8	$\delta_7, \delta_6, \delta_5, \delta_4$	$\delta_3, \delta_1, \delta_8$	$\emptyset$	7	26
9	$\delta_6, \delta_5, \delta_1$	δ_3, δ_4	$\delta_7, \delta_2, \delta_8$	1	25
10	$\delta_6, \delta_4, \delta_1$	δ_3, δ_5	$\delta_7, \delta_2, \delta_8$	1	25

Figure 4.9: The default spaces generated for the example in Figure 4.8.

defaults with equal weights, the ordering is obtained from the indices of the defaults.

Chapter 5

Future Work

We end this report by briefly indicating current research activities which might be integrated into future releases of the FLEX system. These activities mainly address the following issues:

1. test/compute
2. weighted defaults
3. C++ reimplementation
4. graphical user interface

We will integrate descriptions of these extensions in our online-documentation which is available via the FLEX Home Page (http://www.cs.tu-berlin.de/∼flex).

Test/Compute

The current version of the FLEX system already supports the integration of external predicates via the test/compute construct. Though we use these constructs in our own application there are still a number of open issues which have to be worked out in detail. The basic idea of the test/compute construct is to realize functionality not supported by the FLEX representation language as PROLOG predicates [Kortüm 93]. Whereas the test construct is used to test whether an object satisfies certain conditions, the compute construct is used to compute role fillers at an object.

The functionality of compute is illustrated by the following example:

```
 vat   :<   range(number) and feature.
book   =>   compute(vat,book_vat(costs)).
```

This rule says that the value of the feature 'vat' is computed by a PROLOG predicate 'book_vat' which takes the value of the feature 'costs' as argument. Thus one has to define this predicate, e.g. as:

```
book_vat(Vat,[Price]) :–
      Vat is Price * 0.07.
```

Note that the first argument of this predicate is the result, and the other arguments are the filler lists of the roles/features used in the compute construct.

One of the main design problems in connection with these extensions is to decide when and how to trigger their application. Our current strategy is to evaluate test- and compute constructs at objects whose fillers at the relevant roles are closed. Otherwise one would risk to obtain different results, e.g. when evaluating the predicates whenever a new filler is added.

Weighted Defaults

As has been pointed out in Section 3.3, the use of weighted defaults in an application poses several problems. The most important ones are briefly listed below:

1. applying defaults is rather complex;
2. determining appropriate weights is not trivial;
3. understanding and querying default information.

There are several sources of complexity in default reasoning. First note that conflicts can arise between defataults applicable at different objects:

c_1	$\sim m \sim >$	all(r,c_3)
c_2	$\sim n \sim >$	not(c_3)
o_1	::	c_1 and r:o_2
o_2	::	c_2

As a consequence, finding the preferred application of defaults is a non-local task and cannot be performed locally when processing a single object. However, it is not clear whether such interdependencies between defaults applicable at different objects will frequently arise in real applications.

Another source of complexity is the computation of maximal default spaces. In this process object tells ‘o:: c’ are added to a situation and if they are inconsistent, alternatives are generated. In principle, one cannot tell whether two defaults can be jointly applied at an object by looking at the defaults alone—the conflict can arise due to information contained in the defaults *and* information known at the object. In cases, however, where the right-hand sides of defaults are disjoint, one can tell in advance that they will yield a conflict and there is thus no need to test consistency via object tells.

In our current VERBMOBIL application we use defaults to determine the dialogue act of an utterance [Schmitz, Quantz 95]. In this scenario all defaults have a dialogue act as their right-hand side. Since these dialogue acts are mutually disjoint, the defaults can be grouped into equivalence classes, each class containing the defaults which have the same act as conclusion.

In order to determine the dialogue act of an utterance we then compute all applicable defaults, split them into equivalence classes, and take the dialogue act whose equivalence class has the heighest weight. Note that this involves just one object tell and thus avoids the costly computation of alternative situations.

We are currently investigating the theoretical basis of this approach, i.e. the conditions under which this guarantees a complete and sound solution. We thereby hope to obtain insights which help to generalize the approach to a more general setting.

C++ Reimplementation

The developement of “FLEX++”, a rather straight-forward reimplementation of FLEX in C++, was motivated in the beginning of 1995 by resource constraints, posed by an application of FLEX within the field of natural language processing. This application was composed by a large domain model together with exhaustive use of objects and defaults, so that FLEX required considerable time to perform its tasks. As a result, we decided to reimplement FLEX using C++ to achieve a substantial gain in execution speed. Other aspects of the reimplementation are the reduction of memory complexity (to allow FLEX to run on PC-like computers instead of workstations) and the efficient management of secondary storage media (hard disk) to provide persistency.

To achieve these goals, much care has been taken to choose a representation paradigm that takes advantage of the more flexible data structures of C++ (in comparison with Prolog) and that allows for further optimizations of the system behavior. As a result we adopted many of the optimization techniques that proved their efficiency within FLEX and added a few other ones, that became applicable due to the more efficient (fine grain) data structure of C++. These new techniques include:

Incremental Normalization: Generic normalization is avoided in favor of the process of adding information to already given information. Thus multiple application of inference rules is avoided.

Ordering Normal Forms: The internal representation of concepts and objects is ordered with respect to the role hierarchy. This way inheritance of the all, atleast, atmost and fillers constructs is handled in an efficient way.

Local Caches: The very efficient “==0”-comparison of C++ is used to determine the existence or non-existence of cached information at many places within the FLEX++ system. Such comparisons are by far less efficient within Prolog, so that caching did not pay the effort of comparison.

Structure Sharing: Parts of the internal C++ representation are structure-shared. This way expensive copy and compare operations are avoided. Instead simple pointer comparisons are sufficient to determine equality.

In addition to these techniques that are used to optimize the runtime behavior, considerations have been made concerning the frequency of usage of individual parts of the internal representation. As a result it turned out that the internal representation now consists of only 5 different data structures, which are of fixed lenght (with exception of sets, that have to be handled specificly). These fixed-length objects can be stored and retrieved by a conventional relational database. In contrast to this, the variable-length terms of Prolog require special (and less efficent) treatment.

Graphical User Interface

We are currently implementing the FLEX GUI. This interface is implemented using TCL/TK. The interface communicates by TCP/IP with the prolog FLEX process.

The main advantage of graphical user interfaces is to make interaction with a complex system easier. This can be achieved by looking at a system on a higher level of abstraction and by using some techniques like hypertexts, hierarchies and graphical editors. FLEX is a DL system with an expressive term description language. It offers situated object descriptions and defaults.

The FLEX controller window consists of a main menu and a shell. You can type in any command accepted by the FLEX system. If FLEX can not solve a given goal it returns 'no'. In the other case the GUI actualizes its working windows due to the new FLEX system state. Answers are displayed after the excecution of an entered command.

There are currently three types of working windows. The first window type displays TBOX definitions or ABOX assertions as hypertexts. The second one displays concept or role hierarchies in a graphical manner as DAGs. A third type of window allows you to place objects of the ABOX and establish existing role links between these objects.

Every working window (ww) has at least the following properties.

- the standard window operations like moving and scaling windows can be done;

- all operations of the menu FLEX.Windows are applied to ww;

- ww.ww-name.quit closes the ww;
- ww.permissions.focus toggles if an other window can focus some entity in the ww.

TBOX Hypertext windows

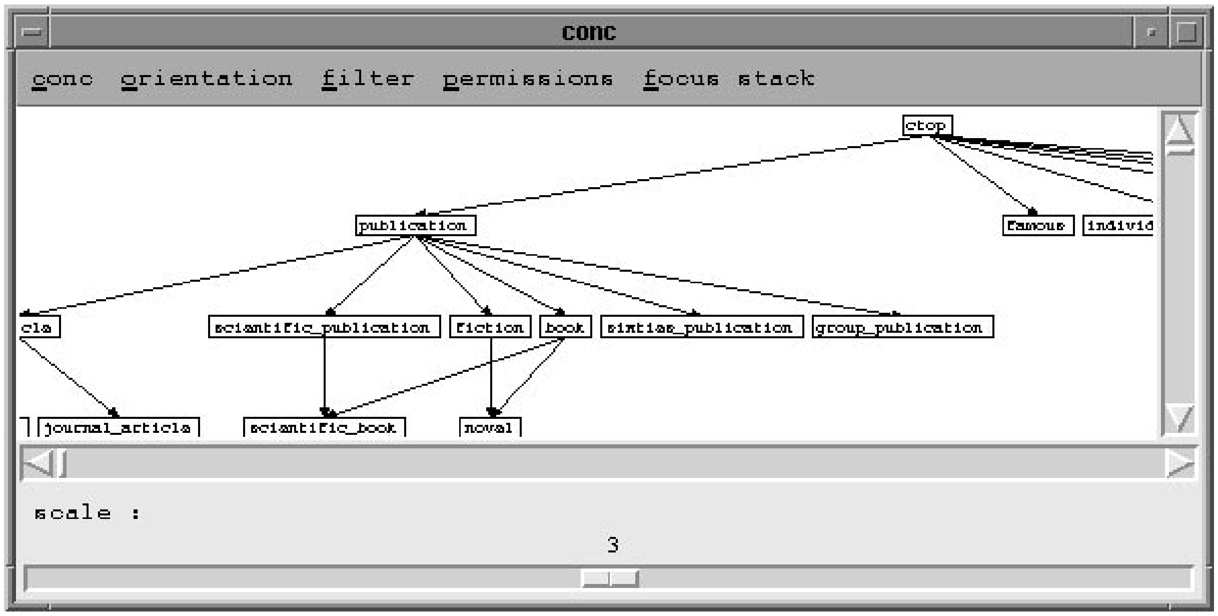
These window type is used to see what definitions have been entered into the system. The order from the top to the bottom is the assertion time. You can scroll and search through these definitions. If you enter a concept, role or object all occurences of this entity will be highlighted. Other working windows may focus definitions if you give local permission to do so.

ABOX Hypertext windows

To open this kind of window you will have to choose a situation from the menu FLEX.Definitions.abox in situation. As a result only object assertions of this situation will be shown.

Concept and Role hierarchie windows

Concept- and Role DAGs can be browsed. You can get Information abount single nodes, hiding subtrees and scaling your view. This is an example how it looks like:



ABOX object-net windows

Objects are displayed as nodes connected by role arrows. It should be possible to produce a relevant view on the object structures. We are investigating automatic procedures to produce these views.

configuration

Configuration panels can be used to browse and modify the activation status of inference rules. Inference rules are odered into classes and offer a description.

help

A HTML based hypertext help window gives information about the GUI itself, the FLEX system commands and syntax. Help files can also be viewed with normal WWW browsers.

References

[Baader, Hollunder 92] F. Baader, B. Hollunder, "Embedding Defaults into Terminological Knowledge Representation Formalisms", *KR-92*, 306–317

[Baader, Hollunder 93] F. Baader, B. Hollunder, *How to Prefer More Specific Defaults in Terminological Default Logic*, *IJCAI-93*, 669–674

[Baader et al. 92] F. Baader, B. Hollunder, B. Nebel, H.J. Profitlich, E. Franconi, "An Empirical Analysis of Optimization Techniques for Terminological Representation Systems", *KR-92*, 270–281

[Barwise, Perry 83] J. Barwise, J. Perry, *Situations and Attitudes*, Cambridge: MIT Press, 1983

[Bergmann, Quantz 95] F.W. Bergmann, J.J. Quantz, "Parallelizing Description Logics", in I. Wachsmuth, C.-R. Rollinger, W. Brauer (eds), *KI-95: Advances in Artificial Intelligence*, Berlin: Springer, 1995, 137–148

[Brachman 77] R.J. Brachman, "What's in a Concept: Structural Foundations for Semantic Networks", *International Journal of Man-Machine Studies* **9**, 127–152, 1977

[Brachman 79] R.J. Brachman, "On the Epistemological Status of Semantic Networks", in N.V. Findler (Ed.), *Associative Networks: Representation and Use of Knowledge by Computers*, New York: Academic Press, 1979, 3–50

[Brachman, Levesque 84] R.J. Brachman, H. Levesque, "The Tractability of Subsumption in Frame-Based Description Languages", *AAAI-84*, 34–37

[Brachman, Schmolze 85] R.J. Brachman, J.G. Schmolze, "An Overview of the KL-ONE Knowledge Representation System" *Cognitive Science* **9**, 171–216, 1985

[Brachman et al. 83] R.J. Brachman, R.E. Fikes, H.J. Levesque, "KRYPTON: A Functional Approach to Knowledge Representation", *IEEE Computer* **16**, 67–73, 1983

[Brachman et al. 91] R. Brachman, D.L. McGuiness, P.F. Patel-Schneider, L. Alperin Resnick, A. Borgida, "Living with CLASSIC: When and How to Use a KL-ONE-like Language", in J. Sowa (Ed.), *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, San Mateo: Morgan Kaufmann, 1991, 401–456

[Carpenter 92] B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge: Cambridge University Press, 1992

[Collins, Quillian 69] A. Collins, M.R. Quillian, "Retrieval Time from Semantic Memory", *Journal of Verbal Learning and Verbal Behavior* **8**, 241–248 1969

[Decio et al. 91] E. Decio, P. Petrin, L. Spampinato, "Pushing the Terminological Barrier", in M. Lenzerini, D. Nardi, M. Simi (eds), *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Chichester: John Wiley, 1991, 113–133

[Donini et al. 91a] F.M. Donini, M. Lenzerini, D. Nardi, W. Nutt, “The Complexity of Concept Languages”, *KR’91*, 151–162

[Donini et al. 91b] F.M. Donini, M. Lenzerini, D. Nardi, W. Nutt, “Tractable Concept Languages” *IJCAI-91*, 458–463

[Donini et al. 92] F.M. Donini, M. Lenzerini, D. Nardi, A. Schaerf, W. Nutt, “Adding Epistemic Operators to Concept Languages”, *KR-92*, 342–353

[Edelmann, Owsnicki 86] J. Edelmann, B. Owsnicki, “Data Models in Knowledge Representation Systems: A Case Study”, in C.-R. Rollinger, W. Horn (eds), *Proceedings of GWAI’86*, Berlin: Springer, 1986, 69–74

[Fischer 92] M. Fischer, *The Integration of Temporal Operators into a Terminological Representation System*, KIT-Report 99, Technische Universität Berlin, 1992

[Franconi 92] E. Franconi, “Collective Entities and Relations in Concept Languages”, in R. MacGregor, D. McGuiness, E. Mays, T. Russ (eds), *AAAI Fall Symposium’92, Issues in Description Logics: Users meet Developers*, 31–35

[Hayes 77] P.J. Hayes, “In Defense of Logic”, *IJCAI-77*, 559–565

[Hayes 80] P.J. Hayes, “The Logic of Frames”, in D. Metzing (Ed.), *Frame Conceptions and Text Understanding*, Berlin: de Gruyter, 1980, 46–61

[Heinsohn 91] J. Heinsohn, “A Hybrid Approach for Modeling Uncertainty in Terminological Logics”, in R. Kruse, P. Siegel (eds), *Symbolic and Quantitative Approaches to Uncertainty, Proceedings of the European Conference EC-SQAU*, Berlin: Springer, 1991, 198–205

[Heinsohn, Owsnicki-Klewe 88] J. Heinsohn, B. Owsnicki-Klewe, “Probabilistic Inheritance and Reasoning in Hybrid Knowledge Representation Systems”, in [Hoeppner 88], 51–60,

[Hoeppner 88] W. Hoeppner (Ed.), *Proceedings of GWAI’88*, Berlin: Springer, 1988

[Hoppe et al. 93] T. Hoppe, C. Kindermann, J.J. Quantz, A. Schmiedel, M. Fischer, BACK V5 *Tutorial & Manual*, KIT Report 100, Technische Universität Berlin, 1993

[Kasper 87] B. Kasper, “A Unification Method for Disjunctive Feature Descriptions”, *ACL-87*, 235–242

[Kindermann 95] C. Kindermann, *Verwaltung assertorischer Inferenzen in terminologischen Wissensbanksystemen*, PhD Thesis, Technische Universität Berlin, 1995

[Kobsa 89] A. Kobsa, *The* SB-ONE *Knowledge Representation Workbench*, 1989

[Kortüm 93] G. Kortüm, *How To Compute 1+1? A Proposal for the Integration of External Functions and Computed Roles into* BACK, KIT Report 103, Technische Universität Berlin, 1993

[MacGregor 91] R. MacGregor, “Using a Description Classifier to Enhance Deductive Inference”, in *Proceedings Seventh IEEE Conference on AI Applications*, 141–147, Miami, Florida, February, 1991

[Minsky 75] M. Minsky, “A Framework for Representing Knowledge”, in P.H. Winston (Ed.), *The Psychology of Computer Vision*, New York: McGraw-Hill, 1975, 211–277

[Nebel 90] B. Nebel, *Reasoning and Revision in Hybrid Representation Systems*, Berlin: Springer, 1990

[Owsnicki-Klewe 88] B. Owsnicki-Klewe, “Configuration as a Consistency Maintenance Task”, in [Hoeppner 88], 77–87

[Patel-Schneider 84] P.F. Patel-Schneider, “Small can be Beautiful in Knowledge Representation”, *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, 11–16, Denver, 1984

[Patel-Schneider 87] P.F. Patel-Schneider, *An Approach to Practical Object-Based Knowledge Representation*, Technical Report 68, Schlumberger Palo Alto Research, 1987

[Quantz 92a] J.J. Quantz, “How to Fit Generalized Quantifiers into Terminological Logics”, *ECAI-92*, 543–547

[Quantz 92b] J.J. Quantz, “A Step Towards Second Order”, in R. MacGregor, D. McGuiness, E. Mays, T. Russ (eds), *AAAI Fall Symposium’92, Issues in Description Logics: Users meet Developers*, 78–82, 1992

[Quantz 93] J.J. Quantz, “Interpretation as Exception Minimization”, *IJCAI-93*, 1310–1315

[Quantz 95] J.J. Quantz, “Preferential Disambiguation in Natural Language Processing”, PhD Thesis, Technische Universität Berlin, 1995

[Quantz, Royer 92] J.J. Quantz, V. Royer, “A Preference Semantics for Defaults in Terminological Logics”, *KR-92*, 294–305

[Quantz, Schmitz 94] J.J. Quantz, B. Schmitz, “Knowledge-Based Disambiguation for Machine Translation”, *Minds and Machines* **4**, 1994, 39–57

[Quantz, Suska 94] J.J. Quantz, S. Suska, “Weighted Defaults in Description Logics—Formal Properties and Proof Theory”, in B. Nebel, L. Dreschler-Fischer (eds), *KI-94: Advances in Artificial Intelligence*, Berlin: Springer, 1994, 178–189

[Quillian 68] M.R, Quillian, “Semantic Memory”, in M. Minsky (Ed.), *Semantic Information Processing*, Cambridge (Mass): MIT Press, 1968, 216–270

[Royer, Quantz 92] V. Royer, J.J. Quantz, “Deriving Inference Rules for Terminological Logics”, in D. Pearce, G. Wagner (eds), *Logics in AI, Proceedings of JELIA’92*, Berlin: Springer, 1992, 84–105

[Royer, Quantz 93] V. Royer, J.J. Quantz, *Deriving Inference Rules for Description Logics: a Rewriting Approach into Sequent Calculi*, KIT Report 111, Technische Universität Berlin, 1993

[Royer, Quantz 94] V. Royer, J.J. Quantz, “On Intuitionistic Query Answering in Description Bases”, in A. Bundy (Ed.), *CADE-94*, Berlin: Springer, 1994, 326–340

[Schild 89] K. Schild, *Towards a Theory of Frames and Rules*, KIT Report 76, Technische Universität Berlin, 1989

[Schild 91] K. Schild, *A Tense-Logical Extension of Terminological Logics*, KIT Report 92, Technische Universität Berlin, 1991

[Schmiedel 90] A. Schmiedel, "A Temporal Terminological Logic", *AAAI'90*, 640–645, 1990

[Schmitz, Quantz 95] B. Schmitz, J.J. Quantz, "Dialogue Acts in Automatic Dialogue Interpreting", *TMI-95*, 33–47

[Schmolze, Israel 83] J. Schmolze, D. Israel, "KL-ONE: Semantics and Classification", in *BBN Annual Report*, Rep.No. 5421, 27–39, 1983

[Schmolze, Mark 91] J. Schmolze, W.S. Mark, "The NIKL Experience", *Computational Intelligence* **6**, 48–69, 1991

[Sundholm 83] G. Sundholm, "Systems of Deduction", in D. Gabbay, F. Guenthner (eds), *Handbook of Philosophical Logic*, Vol. I: Elements of Classical Logic, Dordrecht: Reidel, 1983, 133–188

[Suska 94] S. Suska, *A Proof Theory for Preferential Default Description Logics*, KIT Report 117, Technische Universität Berlin, 1994

[Vilain 85] M.B. Vilain, "The Restricted Language Architecture of a Hybrid Representation System", *IJCAI-85*, 547–551

[Woods 75] W.A. Woods, "What's in A Link: Foundations for Semantic Networks" in D.G. Bobrow, A.M. Collins (eds), *Representation and Understanding: Studies in Cognitive Science*, New York: Academic Press, 1975, 35–82

Appendix A

How To Install the FLEX System

The installation of the FLEX System, the current version is now 1.6, and it's Graphical User Interface (GUI) can be done by following the sequence of instructions below. There are some preconditions before you can install your system properly. This software has been developed on a **Sparc** compatible hardware. As your operating system you should run **Solaris SunOS 5.3**. If you have **Quintus Prolog 3.1.4 or 3.2** installed, this software will also run with older SunOS versions. To use the FLEX GUI you must be able to run X11 Version **X11R6** or Open Windows. Additional online infomation can be found online at http://www.cs.tu-berlin.de/∼flex. Here you can find software updates and lastest news about the system.
To obtain the latest FLEX System [1] package just connect to our FTP server. The following releases will also be available at this site.

```
> ftp ftp.cs.tu-berlin.de
> Name: anonymous
> Password: <enter your email adress>
> cd pub/local/kit/software/FLEX
> get flex1.6.tar.gz
```

Now you are able to create you personal FLEX directory, unpack the system and to install it.

```
> mkdir ~/FLEX                  % only an example directory
> mv flex1.6.tar.gz ~/FLEX
> cd ~/FLEX
> gunzip flex1.6.tar.gz
> tar -xvf flex1.6.tar
> ./install                     % do the installation ...
```

[1] The following describes how to get the current version 1.6

The following directories will be created :

```
BIN/{flex.prog,f} % FLEX-shell (solaris runtime or
                                Quintus saved state)
CONFIG/
GUI/              % tcl source code GUI
PL/               % FLEX prolog code
TESTS/            % FLEX test suite
TMP/
DOCU/             % Documentation
```

Please remove the .tar files if installation was successful and enter the following definitions:

```
setenv FLEX_SYSTEM_PATH ˜/FLEX          % Example
set path=( ˜/FLEX/BIN $path)
setenv LD_LIBRARY_PATH  /usr/X11/lib    % or where ever
                                        % libX11.so.??? is
```

If you use OpenWindows you should set the following:

```
setenv OPENWINHOME /usr/openwin
```

Now you can call the FLEX System with one of these commands.

```
˜/FLEX/BIN/f                        % if you have Quintus
˜/FLEX/BIN/flex.prog                % as stand alone
```

The graphical user interface is called from the FLEX shell. Just type the following command an wait for the GUI main window.

```
| ?- gui.
```

Call the GUI from the FLEX shell :

```
| ?- gui.
```

If nessesary, please modify the first 3 Lines of the Makefile in FLEX/GUI/COMP/ to compile flex_gui_prog' properly.
You can call the main Makefile directly with the following targets :

```
flex:            # target : make flex and dump f
gui:             # target : make graphical user interface
flex_runtime:    # target : make flex stand alone (nearly)
```

Appendix B

FLEX Syntax

Interaction

⟨*interaction*⟩	::=	**flexinit**
	|	**flextell**(⟨*tell-expression*⟩)
	|	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
	|	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]PROLOG-VAR)
	|	**flexstate**(⟨*state*⟩)
	|	**flexread**(⟨*file*-NAME⟩)
	|	**flexdump**(⟨*file*-NAME⟩[,Comment])
	|	**flexload**(⟨*file*-NAME⟩)
⟨*state*⟩	::=	**verbosity** = **silent**
	|	**verbosity** = **error**
	|	**verbosity** = **warnings**
	|	**verbosity** = **info**
	|	**verbosity** = **trace**
	|	**introduction** = **forward**
	|	**introduction** = **noforward**
	|	**class_objects** = **on**
	|	**class_objects** = **off**
	|	**defspace_dbox** = **best**
	|	**defspace_dbox** = **all**
	|	**syntax_check** = **on**
	|	**syntax_check** = **off**

Tell/Ask Expressions

⟨*tell-expression*⟩	::=	⟨*definition*⟩
	|	⟨*rule*⟩
	|	⟨*description*⟩
	|	⟨*default*⟩
	|	⟨*disjointness*⟩
	|	⟨*sit-extension*⟩
	|	⟨*macro-definition*⟩

⟨*definition*⟩	::=	⟨*term*-NAME⟩ := ⟨*term*⟩[**with filter**=⟨*filter-list*⟩]
	|	⟨*term*-NAME⟩ :< ⟨*term*⟩[**with filter**=⟨*filter-list*⟩]
⟨*rule*⟩	::=	⟨*concept*⟩ => ⟨*concept*⟩
⟨*default*⟩	::=	⟨*concept*⟩~⟨INTEGER⟩~> ⟨*concept*⟩
⟨*description*⟩	::=	⟨*object*-NAME⟩ :: ⟨*concept*⟩[**in** ⟨*sit-ref*⟩]
	|	PROLOG-VAR :: ⟨*concept*⟩[**in** ⟨*sit-ref*⟩]
⟨*disjointness*⟩	::=	⟨*concept*-NAME⟩<> ⟨*concept*-NAME⟩
	|	<> '['⟨*concept*-NAME⟩{,⟨*concept*-NAME⟩}*']'
⟨*sit-extension*⟩	::=	⟨*sit*-NAME⟩<<= ⟨*sit-ref*⟩
⟨*macro-definition*⟩	::=	⟨*macro*⟩ $*$= ⟨*term*⟩
⟨*macro*⟩	::=	⟨*macro*-NAME⟩[(PROLOG-VAR{,PROLOG-VAR}*)]
⟨*ask-expression*⟩	::=	⟨*term*⟩ ?< ⟨*term*⟩
	|	⟨*object*-NAME⟩ ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩]
	|	PROLOG-VAR ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩]
	|	**disjoint**(⟨*term*⟩,⟨*term*⟩)
	|	**subsumes**(⟨*term*⟩ ⟨*term*⟩)
	|	**equivalent**(⟨*term*⟩,⟨*term*⟩)
	|	**incoherent**(⟨*term*⟩)
	|	**satisifes**(⟨*term*-NAME⟩,⟨*filter-list*⟩)

Terms

```
  ⟨entity⟩   ::=  ⟨term⟩
               |  ⟨value⟩

    ⟨term⟩   ::=  ⟨concept⟩
               |  ⟨role⟩

   ⟨value⟩   ::=  ⟨object-NAME⟩
               |  ⟨number-INSTANCE⟩
               |  ⟨string-INSTANCE⟩
```

Concept Terms

```
 ⟨concept⟩   ::=  ⟨concept-NAME⟩
               |  ⟨number⟩
               |  string
               |  ⟨macro-NAME⟩
               |  ctop | anything
               |  cbot | nothing
               |  prim(⟨concept-NAME⟩)
               |  ⟨concept⟩ and ⟨concept⟩
               |  ⟨concept⟩ or ⟨concept⟩
               |  not(⟨concept⟩)
               |  all(⟨role⟩,⟨concept⟩)
               |  the(⟨role⟩,⟨concept⟩)
               |  some(⟨role⟩[,⟨concept⟩])
               |  no(⟨role⟩[,⟨concept⟩])
               |  atleast(⟨INTEGER⟩,⟨role⟩[,⟨concept⟩])
               |  atmost(⟨INTEGER⟩,⟨role⟩[,⟨concept⟩])
               |  exactly(⟨INTEGER⟩,⟨role⟩[,⟨concept⟩])
               |  ⟨role⟩:⟨value⟩  | ⟨role⟩:‘[’⟨value⟩{,⟨value⟩}*‘]’
               |  ⟨role⟩:⟨term⟩
               |  rvm_equal(⟨role⟩,⟨role⟩)
               |  oneof(‘[’⟨object-NAME⟩{,⟨object-NAME⟩}*‘]’)
               |  rvm_inst(⟨role⟩,⟨role⟩)
```

Role Terms

⟨*role*⟩	::=	⟨*role*-NAME⟩
	|	**rtop** | **anyrole**
	|	**rbot** | **nothing**
	|	⟨*role*⟩ **and** ⟨*role*⟩
	|	⟨*role*⟩ **or** ⟨*role*⟩
	|	**not**(⟨*role*⟩)
	|	**prim**(⟨*role*-NAME⟩)
	|	**domain**(⟨*concept*⟩)
	|	**range**(⟨*concept*⟩)
	|	**inv**(⟨*role*⟩)
	|	⟨*role*⟩ **comp** ⟨*role*⟩
	|	**feature**
	|	**term_valued**

Number Terms

⟨*number*⟩	::=	**number**
	|	⟨*number*-NAME⟩
	|	⟨*number-range*⟩
	|	⟨*number*-INSTANCE⟩
⟨*number-range*⟩	::=	⟨*lower-limit*⟩
	|	⟨*upper-limit*⟩
	|	⟨*number*-INSTANCE⟩.. ⟨*number*-INSTANCE⟩
⟨*lower-limit*⟩	::=	**gt**(⟨*number*-INSTANCE⟩)
	|	**ge**(⟨*number*-INSTANCE⟩)
⟨*upper-limit*⟩	::=	**lt**(⟨*number*-INSTANCE⟩)
	|	**le**(⟨*number*-INSTANCE⟩)

Methods

⟨*method*⟩	::=	**all**(⟨*role*⟩[,⟨*sit*-NAME⟩])
	|	**atleast**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])
	|	**atmost**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])
	|	**concepts**(⟨*sit*-NAME⟩)
	|	**dir_subs**
	|	**dir_supers**
	|	**domain**
	|	**equivalents**
	|	**fillers**(⟨*role*⟩[,⟨*sit*-NAME⟩])
	|	**filter**
	|	**help**
	|	**instances**(⟨*sit*-NAME⟩)
	|	**msc**(⟨*sit*-NAME⟩)
	|	**range**
	|	**subs**
	|	**supers**
	|	**tvf_filler**(⟨*role*⟩[,⟨*sit*-NAME⟩])
⟨*options*⟩	::=	**box**=⟨*box*⟩
	|	**filter**=⟨*filter-list*⟩
⟨*box*⟩	::=	rules
	|	defaults
⟨*filter-list*⟩	::=	⟨*filter*-NAME⟩
	|	‘[’⟨*filter*-NAME⟩{,⟨*filter*-NAME⟩}*‘]’
⟨*sit-ref*⟩	::=	⟨*sit*-NAME⟩
	|	PROLOG-VAR

Appendix C

FLEX Semantics

In this chapter we specify a formal semantics for FLEX. We begin by specifying the semantics of the basic DL, then specify a preferential semantics for weighted defaults, and finally present a proof theory for weighted defaults. The presentation is largely based on [Quantz 95].

C.1 The Basic DL

For the specification of a formal semantics we use a formal syntax, as specified below:

$$\begin{array}{lcl}
\mathrm{c} & ::= & \top \mid \bot \mid \mathrm{c}_p \mid \mathrm{c}_n \mid \neg\, \mathrm{c} \mid \mathrm{c}_1 \sqcap \mathrm{c}_2 \mid \mathrm{c}_1 \sqcup \mathrm{c}_2 \mid \mathbf{k}\mathrm{c} \mid \mathrm{f}{\in}\mathrm{f}_c \\
 & \mid & \geq n\ \mathrm{r{:}c} \mid \leq n\ \mathrm{r{:}c} \mid \forall \mathrm{r{:}c} \mid \mathrm{r}_1{=}\mathrm{r}_2 \mid \mathrm{r{:}o} \mid \{\mathrm{o}_1, \ldots, \mathrm{o}_m\}^{\vee} \\
 & \mid & \mathrm{f}_c\mathrm{:c} \mid \mathrm{f}_r\mathrm{:r} \mid \top^{\mathrm{n}} \mid \mathrm{n} \mid > n \mid \geq n \mid < n \mid \leq n \mid n_1..n_2 \\
\mathrm{r} & ::= & \top^{\mathrm{r}} \mid \bot^{\mathrm{r}} \mid \mathrm{r}_p \mid \mathrm{r}_n \mid \neg\, \mathrm{r} \mid \mathrm{r}_1 \sqcap \mathrm{r}_2 \mid \mathrm{r}_1 \sqcup \mathrm{r}_2 \mid \mathbf{k}\mathrm{r} \\
 & \mid & {}_{\mathrm{c}}|\mathrm{r} \mid \mathrm{r}|_{\mathrm{c}} \mid \mathrm{r}_1.\mathrm{r}_2 \mid \mathrm{r}^{-} \\
\gamma & ::= & \mathrm{c}_1 \sqsubseteq \mathrm{c}_2 \mid \mathrm{r}_1 \sqsubseteq \mathrm{r}_2 \mid \mathrm{o :: c\ in\ s}, \mathrm{s}_1 \leq \mathrm{s}_2
\end{array}$$

Note that this syntax implicitly assumes the existence of basic sets from which constants such as c_p, f_c, r_n, s_1, or o_m are taken. The following definition makes these sets explicit.

Definition 1
A DL **alphabet** *is a 10-tuple* $A = \langle \mathcal{C}_p, \mathcal{C}_n, \mathcal{R}_p, \mathcal{R}_n, \mathcal{F}_p, \mathcal{F}_n, \mathcal{F}_p^c, \mathcal{F}_p^r, \mathcal{O}, \mathcal{S} \rangle$, *where*

$\mathcal{C}_p$	*is a finite set of primitive concepts*
$\mathcal{C}_n$	*is a finite set of concept names*
$\mathcal{R}_p$	*is a finite set of primitive roles*
$\mathcal{R}_n$	*is a finite set of role names*
$\mathcal{F}_p$	*is a finite set of primitive features*
$\mathcal{F}_n$	*is a finite set of feature names*
$\mathcal{F}^c_p$	*is a finite set of primitive concept-valued features*
$\mathcal{F}^r_p$	*is a finite set of primitive role-valued features*
$\mathcal{O}$	*is a finite set of objects*
$\mathcal{S}$	*is a finite set of situtations*

Given such an alphabet and the syntax of the DL, sets of concepts, roles, infons, propositions,and formulae are implicitly determined. The following definition contains an explicit inductive characterization of terms.

Definition 2 *Let* $A = \langle \mathcal{C}_p, \mathcal{C}_n, \mathcal{R}_p, \mathcal{R}_n, \mathcal{F}_p, \mathcal{F}_n, \mathcal{F}^c_p, \mathcal{F}^r_p, \mathcal{O}, \mathcal{S} \rangle$ *be a* DL *alphabet. The set of* **concepts** $\mathcal{C}$*, the set of* **concept expressions** $\mathcal{C}_e$*, the set of* **roles** $\mathcal{R}$*, the set of* **features** $\mathcal{F}$*, the set of* **concept-valued features** $\mathcal{F}^c$*, and the set of* **role-valued features** $\mathcal{F}^r$*, are inductively defined as the smallest sets satisfying the following constraints:*

1.		$\mathcal{C}_n \cup \mathcal{C}_p$	$\subseteq$	$\mathcal{C}$	$\mathcal{R}_n \cup \mathcal{R}_p \cup \mathcal{F}$	$\subseteq$	$\mathcal{R}$	$\mathcal{F}_n \cup \mathcal{F}_p$	$\subseteq$	$\mathcal{F}$
								$\mathcal{F}^c_p$	$\subseteq$	$\mathcal{F}^c$
								$\mathcal{F}^r_p$	$\subseteq$	$\mathcal{F}^r$
2.		$N_0 \cup \{\top, \bot, \top\mathrm{n}\}$	$\subseteq$	$\mathcal{C}$	$\{\top\mathrm{r}, \bot\mathrm{r}\}$	$\subseteq$	$\mathcal{R}$			
3.	*If*	$o, o_1, \ldots, o_m$	$\in$	$\mathcal{O}$	n, n_1, n_2	$\in$	N_0	f, f_1, f_2	$\in$	$\mathcal{F}$
		c, c_1, c_2	$\in$	$\mathcal{C}$	r, r_1, r_2	$\in$	$\mathcal{R}$	f_c	$\in$	$\mathcal{F}^c$
		c	$\in$	$\mathcal{C}$				f_r	$\in$	$\mathcal{F}^r$
	then									
		$\neg c$	$\in$	$\mathcal{C}$	$\neg r$	$\in$	$\mathcal{R}$			
		$c_1 \sqcap c_2$	$\in$	$\mathcal{C}$	$r_1 \sqcap r_2$	$\in$	$\mathcal{R}$	$f \sqcap r$	$\in$	$\mathcal{F}$
		$c_1 \sqcup c_2$	$\in$	$\mathcal{C}$	$r_1 \sqcup r_2$	$\in$	$\mathcal{R}$			
		$\forall r{:}c$	$\in$	$\mathcal{C}$	$c \vert r$	$\in$	$\mathcal{R}$	$c \vert f$	$\in$	$\mathcal{F}$
		$\geq n\ r{:}c$	$\in$	$\mathcal{C}$	$r \vert c$	$\in$	$\mathcal{R}$	$f \vert c$	$\in$	$\mathcal{F}$
		$\leq n\ r{:}c$	$\in$	$\mathcal{C}$	$r_1.r_2$	$\in$	$\mathcal{R}$	$f_1.f_2$	$\in$	$\mathcal{F}$
		$r{:}o$	$\in$	$\mathcal{C}$	r^-	$\in$	$\mathcal{R}$			
		$r_1 = r_2$	$\in$	$\mathcal{C}$				$c \vert f_c$	$\in$	$\mathcal{F}^c$
		$\{o_1, \ldots, o_m\}^\vee$	$\in$	$\mathcal{C}$				$c \vert f_r$	$\in$	$\mathcal{F}^r$
		$f_c{:}c$	$\in$	$\mathcal{C}$				$f.f_c$	$\in$	$\mathcal{F}^c$
		$f_r{:}r$	$\in$	$\mathcal{C}$				$f.f_r$	$\in$	$\mathcal{F}^r$
		$f \in f_c$	$\in$	$\mathcal{C}$						
		$\{> n, \geq n\}$	$\subseteq$	$\mathcal{C}$						
		$\{< n, \leq n\}$	$\subseteq$	$\mathcal{C}$						
		$n_1..n_2$	$\in$	$\mathcal{C}$						

The set of **epistemic concepts** $\mathcal{C}_k$*, the set of* **epistemic roles** $\mathcal{R}_k$*, and the set of* **epistemic features** $\mathcal{F}_k$ *are inductively defined as the smallest sets satisfying the constraints obtained by substituting* $\mathcal{C}$ *by* $\mathcal{C}_k$*,* $\mathcal{R}$ *by* $\mathcal{R}_k$*, and* $\mathcal{F}$ *by* $\mathcal{F}_k$ *in the above constraints and adding* $\mathrm{k}c \in \mathcal{C}_k$, $\mathrm{k}r \in \mathcal{R}_k$, $\mathrm{k}f \in \mathcal{F}_k$ *in the 'then' part of the third condition.*

Note that according to these definitions the features are a subset of the roles ($\mathcal{F} \subseteq \mathcal{R}$) and nonepistemic terms are a subset of epistemic terms (e.g. $\mathcal{C} \subseteq \mathcal{C}_k$).

The distinctions made between these sets will be used in the definition of different sets of formulae.

Definition 3 *Let* $A = \langle \mathcal{C}_p, \mathcal{C}_n, \mathcal{R}_p, \mathcal{R}_n, \mathcal{F}_p, \mathcal{F}_n, \mathcal{F}_p^c, \mathcal{F}_p^r, \mathcal{O}, \mathcal{S} \rangle$ *be a* DL *alphabet determining the sets* $\mathcal{C}$*,* $\mathcal{C}_k$*, and* $\mathcal{R}$*. The sets of* **nonepistemic object descriptions** *(assertional formulae)* Φ_α*,* **epistemic object descriptions** Φ_κ*,* **definitions** Φ_δ*,* **rules** Φ_ρ*,* **situation extensions** Φ_σ*, and* **subsumptions** *(terminologial formulae)* Φ_θ *are defined as follows:*

$$\begin{aligned}
\Phi_\alpha &\overset{\mathrm{def}}{=} \{o :: c \mathit{\ in\ } s : o \in \mathcal{O}, c \in \mathcal{C}, s \in \mathcal{S}\} \\
\Phi_\kappa &\overset{\mathrm{def}}{=} \{o :: c \mathit{\ in\ } s : o \in \mathcal{O}, c \in \mathcal{C}_k, s \in \mathcal{S}\} \\
\Phi_\delta &\overset{\mathrm{def}}{=} \{c_n \doteq c : c_n \in \mathcal{C}_n, c \in \mathcal{C}\} \cup \\
&\quad\ \{r_n \doteq r : r_n \in \mathcal{R}_n, r \in \mathcal{R}\} \cup \\
&\quad\ \{f_n \doteq f : f_n \in \mathcal{F}_n, f \in \mathcal{F}\} \\
\Phi_\rho &\overset{\mathrm{def}}{=} \{c_1 \Rightarrow c_2 : c_1, c_2 \in \mathcal{C}\} \\
\Phi_\sigma &\overset{\mathrm{def}}{=} \{s_1 \leq s_2 : s_1, s_2 \in \mathcal{S}\} \\
\Phi_\theta &\overset{\mathrm{def}}{=} \{c_1 \sqsubseteq c_2 : c_1, c_2 \in \mathcal{C}\} \cup \\
&\quad\ \{r_1 \sqsubseteq r_2 : r_1, r_2 \in \mathcal{R}\} \cup \\
&\quad\ \{f_1 \sqsubseteq f_2 : f_1, f_2 \in \mathcal{F}\}
\end{aligned}$$

This distinction between different types of formulae is useful since we will allow different types of formulae for DL tells and DL queries, respectively.

Definition 4 *A* DL **knowledge base** *is a pair* $\mathcal{K} = \langle A, \Gamma \rangle$*, where* A *is a* DL *alphabet and* Γ *is a* DL **modeling**, *i.e. a subset of* $\Phi_\alpha \cup \Phi_\delta \cup \Phi_\rho \cup \Phi_\sigma$*. Given such a modeling we define*

$$\begin{aligned}
\Gamma_\alpha &\overset{\mathrm{def}}{=} \Gamma \cap \Phi_\alpha \\
\Gamma_\delta &\overset{\mathrm{def}}{=} \Gamma \cap \Phi_\delta \\
\Gamma_\rho &\overset{\mathrm{def}}{=} \Gamma \cap \Phi_\rho \\
\Gamma_\sigma &\overset{\mathrm{def}}{=} \Gamma \cap \Phi_\sigma
\end{aligned}$$

Γ_δ *is called the* **terminology** *and must satisfy the following two conditions:*

1. *For each $t_n \in \mathcal{C}_n \cup \mathcal{R}_n \cup \mathcal{F}_n$ there is exactly one definition $t_n \doteq t \in \Gamma_\delta$.*
2. *the definitions in Γ_δ can be totally ordered such that for each t_n' occurring on the right-hand side of a definition $t_n \doteq t$, we have $t_n' \doteq t' < t_n \doteq t$.*

These conditions guarantee non-cyclicity of the terminology.

So far we have only dealt with syntactic characterizations. We will now define the semantics of the core DL.

Definition 5 *Let A be a* DL *alphabet. A* **model** *over A is a three-tuple $M = \langle D, \mathcal{I}, \mathcal{W}\rangle$, where D is a set called the* **domain**, *$\mathcal{W}$ is a set of* **interpretation functions**, *and $\mathcal{I} \in \mathcal{W}$. If $\mathcal{J}$ is an interpretation function it satisifies the following constraints:*

$$\begin{array}{rcl}
o \in \mathcal{O} & \Rightarrow & [\![o]\!]^{\mathcal{J}} \in D \\
[\![o_1]\!]^{\mathcal{J}} = [\![o_2]\!]^{\mathcal{J}} & \Rightarrow & o_1 = o_2 \\
c_p \in \mathcal{C}_p, s \in \mathcal{S} & \Rightarrow & [\![c_p, s]\!]^{\mathcal{J}} \subseteq D \\
r_p \in \mathcal{R}_p, s \in \mathcal{S} & \Rightarrow & [\![r_p, s]\!]^{\mathcal{J}} \subseteq D \times D \\
f_p \in \mathcal{F}_p, s \in \mathcal{S} & \Rightarrow & [\![f_p, s]\!]^{\mathcal{J}} \subseteq D \times D \\
\langle d_1, d_2\rangle \in [\![f_p, s]\!]^{\mathcal{J}}, \langle d_1, d_3\rangle \in [\![f_p, s]\!]^{\mathcal{J}} & \Rightarrow & d_2 = d_3 \\
f^c{}_p \in \mathcal{F}^c_p, s \in \mathcal{S} & \Rightarrow & [\![f_c, s]\!]^{\mathcal{J}} \subseteq D \times (2^D \setminus \emptyset) \\
\langle d_1, A_1\rangle \in [\![f_c, s]\!]^{\mathcal{J}}, \langle d_1, A_2\rangle \in [\![f_c, s]\!]^{\mathcal{J}} & \Rightarrow & A_1 = A_2 \\
f^r{}_p \in \mathcal{F}^r_p, s \in \mathcal{S} & \Rightarrow & [\![f_r, s]\!]^{\mathcal{J}} \subseteq D \times (2^{D \times D} \setminus \emptyset) \\
\langle d_1, B_1\rangle \in [\![f_r, s]\!]^{\mathcal{J}}, \langle d_1, B_2\rangle \in [\![f_r, s]\!]^{\mathcal{J}} & \Rightarrow & B_1 = B_2
\end{array}$$

Furthermore, if $\mathcal{J}$ is an interpretation function, $\mathcal{W}$ is a set of interpretation functions, $c, c_1, c_2 \in \mathcal{C}_k$, $c_p \in \mathcal{C}_p$, $r, r_1, r_2 \in \mathcal{R}_k$, $r_p \in \mathcal{R}_p$, $f_p \in \mathcal{F}_p$, $f_c \in \mathcal{F}^c$, $f^c{}_p \in \mathcal{F}^c_p$, $f_r \in \mathcal{F}^r$, $f^r{}_p \in \mathcal{F}^r_p$, $o, o_1, ..., o_m \in \mathcal{O}$, $s \in \mathcal{S}$, $n \in N_0$, $d, d_1, d_2 \in D$ and given the following abbreviations:

$$\begin{array}{rcl}
[\![r, s]\!]^{\mathcal{J},\mathcal{W}}(d_1) & \overset{\text{def}}{=} & \{d_2 \in D : \langle d_1, d_2\rangle \in [\![r, s]\!]^{\mathcal{J},\mathcal{W}}\} \\
[\![f_c, s]\!]^{\mathcal{J},\mathcal{W}}(d) & \overset{\text{def}}{=} & \textit{the set } A \subseteq D \textit{ such that } \langle d, A\rangle \in [\![f_c, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![f_r, s]\!]^{\mathcal{J},\mathcal{W}}(d) & \overset{\text{def}}{=} & \textit{the set } B \subseteq (D \times D) \textit{ such that } \langle d, B\rangle \in [\![f_r, s]\!]^{\mathcal{J},\mathcal{W}}
\end{array}$$

then the following constraints have to be satisfied:

$$\begin{array}{rcl}
[\![\top]\!]^{\mathcal{J},\mathcal{W}} & = & D \\
[\![\bot]\!]^{\mathcal{J},\mathcal{W}} & = & \emptyset \\
[\![c_p, s]\!]^{\mathcal{J},\mathcal{W}} & = & [\![c_p, s]\!]^{\mathcal{J}} \\
[\![\neg c, s]\!]^{\mathcal{J},\mathcal{W}} & = & D \setminus [\![c, s]\!]^{\mathcal{J},\mathcal{W}}
\end{array}$$

$$
\begin{aligned}
[\![c_1 \sqcap c_2, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![c_1, s]\!]^{\mathcal{J},\mathcal{W}} \cap [\![c_2, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![c_1 \sqcup c_2, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![c_1, s]\!]^{\mathcal{J},\mathcal{W}} \cup [\![c_2, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![\mathbf{k}c, s]\!]^{\mathcal{J},\mathcal{W}} &= \bigcap_{\mathcal{J}' \in \mathcal{W}} [\![c, s]\!]^{\mathcal{J}',\mathcal{W}} \\
[\![\geq n\ r{:}c, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : |[\![r, s]\!]^{\mathcal{J},\mathcal{W}}(d) \cap [\![c, s]\!]^{\mathcal{J},\mathcal{W}}| \geq n\} \\
[\![\leq n\ r{:}c, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : |[\![r, s]\!]^{\mathcal{J},\mathcal{W}}(d) \cap [\![c, s]\!]^{\mathcal{J},\mathcal{W}}| \leq n\} \\
[\![\forall r{:}c, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![r, s]\!]^{\mathcal{J},\mathcal{W}}(d) \subseteq [\![c, s]\!]^{\mathcal{J},\mathcal{W}}\} \\
[\![r_1 = r_2, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![r_1, s]\!]^{\mathcal{J},\mathcal{W}}(d) = [\![r_1, s]\!]^{\mathcal{J},\mathcal{W}}(d)\} \\
[\![f \in f_c, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![f, s]\!]^{\mathcal{J},\mathcal{W}}(d) \subseteq [\![f_c, s]\!]^{\mathcal{J},\mathcal{W}}(d)\} \\
[\![r{:}o, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![o]\!]^{\mathcal{J},\mathcal{W}} \in [\![r, s]\!]^{\mathcal{J},\mathcal{W}}(d)\} \\
[\![\{o_1, \ldots, o_m\}^{\vee}, s]\!]^{\mathcal{J},\mathcal{W}} &= \{[\![o_1]\!]^{\mathcal{J}}, \ldots, [\![o_m]\!]^{\mathcal{J}}\} \\
[\![f_3{:}c, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![f_c, s]\!]^{\mathcal{J},\mathcal{W}}(d) \subseteq [\![c, s]\!]^{\mathcal{J},\mathcal{W}}\} \\
[\![f_r{:}r, s]\!]^{\mathcal{J},\mathcal{W}} &= \{d : [\![f_r, s]\!]^{\mathcal{J},\mathcal{W}}(d) \subseteq [\![r, s]\!]^{\mathcal{J},\mathcal{W}}\} \\
[\![\top^{\mathrm{n}}, s]\!]^{\mathcal{J},\mathcal{W}} &= N_0 \\
[\![n, s]\!]^{\mathcal{J},\mathcal{W}} &= n \\
[\![> n, s]\!]^{\mathcal{J},\mathcal{W}} &= \{m \in N_0 : m > n\} \\
[\![\geq n, s]\!]^{\mathcal{J},\mathcal{W}} &= \{m \in N_0 : m \geq n\} \\
[\![< n, s]\!]^{\mathcal{J},\mathcal{W}} &= \{m \in N_0 : m < n\} \\
[\![\leq n, s]\!]^{\mathcal{J},\mathcal{W}} &= \{m \in N_0 : m \leq n\} \\
[\![n_1..n_2, s]\!]^{\mathcal{J},\mathcal{W}} &= \{m \in N_0 : n_1 \leq m \leq n_2\} \\
[\![\top^{\mathrm{r}}, s]\!]^{\mathcal{J},\mathcal{W}} &= D \times D \\
[\![\bot^{\mathrm{r}}, s]\!]^{\mathcal{J},\mathcal{W}} &= \emptyset \\
[\![r_p, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r_p, s]\!]^{\mathcal{J}} \\
[\![\neg r, s]\!]^{\mathcal{J},\mathcal{W}} &= (D \times D) \setminus [\![r, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![r_1 \sqcap r_2, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r_1, s]\!]^{\mathcal{J},\mathcal{W}} \cap [\![r_2, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![r_1 \sqcup r_2, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r_1, s]\!]^{\mathcal{J},\mathcal{W}} \cup [\![r_2, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![\mathbf{k}r, s]\!]^{\mathcal{J},\mathcal{W}} &= \bigcap_{\mathcal{J}' \in \mathcal{W}} [\![r, s]\!]^{\mathcal{J}',\mathcal{W}} \\
[\![c|r, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r, s]\!]^{\mathcal{J},\mathcal{W}} \cap ([\![c, s]\!]^{\mathcal{J},\mathcal{W}} \times D) \\
[\![r|c, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r, s]\!]^{\mathcal{J},\mathcal{W}} \cap (D \times [\![c, s]\!]^{\mathcal{J},\mathcal{W}}) \\
[\![r_1.r_2, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![r_1, s]\!]^{\mathcal{J},\mathcal{W}} \circ [\![r_2, s]\!]^{\mathcal{J},\mathcal{W}} \\
[\![r^{-}, s]\!]^{\mathcal{J},\mathcal{W}} &= \{\langle d_1, d_2 \rangle : \langle d_2, d_1 \rangle \in [\![r, s]\!]^{\mathcal{J},\mathcal{W}}\} \\
[\![f_p, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![f_p, s]\!]^{\mathcal{J}} \\
[\![f^{c}{}_p, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![f^{c}{}_p, s]\!]^{\mathcal{J}} \\
[\![f^{r}{}_p, s]\!]^{\mathcal{J},\mathcal{W}} &= [\![f^{r}{}_p, s]\!]^{\mathcal{J}}
\end{aligned}
$$

Finally we have the constraint

$$\forall \mathcal{I}, \mathcal{J} \in \mathcal{W}, \forall o \in \mathcal{O} \qquad [\![o]\!]^{\mathcal{I}} = [\![o]\!]^{\mathcal{J}}$$

After all these preliminaries we can now define satisfaction of a knowledge base by a model:

Definition 6 *Let $\mathcal{K} = \langle A, \Gamma \rangle$ be a knowledge base and $M = \langle D, \mathcal{I}, \mathcal{W} \rangle$ a model over A. M* **is a model of** Γ *($M \models^k \Gamma$) iff $\forall \mathcal{J} \in \mathcal{W}$*

$$\begin{array}{rl} [\![o]\!]^{\mathcal{J}} \in [\![c, s]\!]^{\mathcal{J},\mathcal{W}} & \forall o :: c \; in \; s \in \Gamma_\alpha \\ [\![t_n, s]\!]^{\mathcal{J},\mathcal{W}} = [\![t, s]\!]^{\mathcal{J},\mathcal{W}} & \forall t_n \doteq t \in \Gamma_\delta, \forall s \in \mathcal{S} \\ [\![\mathbf{k}c_1, s]\!]^{\mathcal{J},\mathcal{W}} \subseteq [\![\mathbf{k}c_2, s]\!]^{\mathcal{J},\mathcal{W}} & \forall c_1 \Rightarrow c_2 \in \Gamma_\rho, \forall s \in \mathcal{S} \\ [\![\mathbf{k}c, s_1]\!]^{\mathcal{J},\mathcal{W}} \subseteq [\![\mathbf{k}c, s_2]\!]^{\mathcal{J},\mathcal{W}} & \forall s_1 \leq s_2 \in \Gamma_\sigma, \forall c \in \mathcal{C} \end{array}$$

In the following we will write $M \models^k o :: c \; in \; s$ (for $o :: c \; in \; s \in \Phi_\kappa$) iff $\forall \mathcal{J} \in \mathcal{W} [\![o]\!]^{\mathcal{J}} \in [\![c, s]\!]^{\mathcal{J},\mathcal{W}}$.

On the basis of this definition we can already define entailment of nonepistemic queries:

Definition 7 *Let $\mathcal{K} = \langle A, \Gamma \rangle$ be a knowledge base, $o :: c \; in \; s \in \Phi_\alpha$, and $t_1 \sqsubseteq t_2 \in \Phi_\theta$:*

$$\begin{array}{rll} \Gamma \models o :: c \; in \; s & \textit{iff} & \textit{in every model } M \textit{ of } \Gamma \textit{ we have } M \models^k o :: c \; in \; s \\ \Gamma \models t_1 \sqsubseteq t_2 & \textit{iff} & \textit{in every model } M \textit{ of } \Gamma \forall s \in \mathcal{S} \; [\![t_1, s]\!]^{\mathcal{I},\mathcal{W}} \subseteq [\![t_2, s]\!]^{\mathcal{I},\mathcal{W}} \end{array}$$

Note that for defining entailment of epistemic queries we need to ensure maximality of $\mathcal{W}$. For reasons that will become obvious below we will define maximality by using techniques from preferential model theory.

C.2 A Preferential Semantics for Weighted Defaults

In this section we will present a Preferential Default Description Logic (PDDL) based on weighted defaults. This PDDL differs from the one developed in [Quantz, Suska 94] in two respects:

1. the semantics is modified to take into account the situated descriptions used in the core DL.

2. the **k** operator is used to weaken the semantics in such a way that weighted defaults behave as forward-chaining trigger rules instead of as material implications.

Whereas the first modification has no impact on the formal properties of the resulting preferential entailment relation, the second modification "destroys" formal properties as *Or* or *Rational Monotonicity*. The advantage of the weakened semantics is that it yields a proof theory which has better computational properties in the average case.

The basic idea of preferential semantics is to use orderings on models for the definition of nonmonotonic preferential entailment relations. Whereas strict information divides the set of models into two subsets, namely those models satisfying the information and those not satisfying it, default information is used to establish a fine grained ordering on models.

We begin by defining entailment of epistemic descriptions:

Definition 8 *Let A be a* DL *alphabet and $M = \langle D_1, \mathcal{I}, \mathcal{W}_1 \rangle$, $N = \langle D_2, \mathcal{J}, \mathcal{W}_2 \rangle$ two models over A. N is* **epistemically preferred over** *M ($M \sqsubset^k N$) iff*

1. $D_1 = D_2$
2. $\mathcal{I} = \mathcal{J}$
3. $\mathcal{W}_1 \subset \mathcal{W}_2$

Given this ordering on models we define $\sqsubset^k$-maximal models for knowledge bases:

Definition 9 *Let $\mathcal{K} = \langle A, \Gamma \rangle$ be a knowledge base and M a model over A. M is a* **$\sqsubset^k$-maximal** *model of Γ iff*

1. *M is a model of Γ and*
2. *there is no model N of Γ with $M \sqsubset^k N$*

Entailment of epistemic descriptions is then defined by taking into account only the $\sqsubset^k$-maximal models:

Definition 10 *Let $\mathcal{K} = \langle A, \Gamma \rangle$ be a knowledge base and $\gamma \in \Phi_\kappa$.*

$$\Gamma \mathrel{|\!\!\approx^k} \gamma \quad \textit{iff} \quad \textit{for all } \sqsubset^k\textit{-maximal models } M \textit{ of } \Gamma \textit{ we have } M \models^k \gamma$$

The effect of $\sqsubset^k$ is easily illustrated by considering the following example:

$$\begin{array}{rcl} M_1 & = & \langle \{d_1, d_2\}, \mathcal{I}, \{\mathcal{I}\} \rangle \\ M_2 & = & \langle \{d_1, d_2\}, \mathcal{I}, \{\mathcal{I}, \mathcal{J}\} \rangle \\ [\![\mathrm{o}_1]\!]^{\mathcal{I}} & = & d_1 \\ [\![\mathrm{o}_2]\!]^{\mathcal{I}} & = & d_2 \\ [\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{I}} & = & \{\langle d_1, d_2 \rangle, \langle d_1, d_1 \rangle\} \end{array}$$

$$\begin{array}{rcl}
[\![\mathrm{o}_1]\!]^{\mathcal{J}} & = & d_1 \\
[\![\mathrm{o}_2]\!]^{\mathcal{J}} & = & d_2 \\
[\![\mathrm{r},\mathrm{s}]\!]^{\mathcal{J}} & = & \{\langle d_1, d_2\rangle\} \\
M_1 & \sqsubset^k & M_2 \\
M_1 & \not\models^k & \mathrm{o}_1 :: \leq 1\ \mathbf{k}\mathrm{r}\ \mathrm{in}\ \mathrm{s} \\
M_2 & \models^k & \mathrm{o}_1 :: \leq 1\ \mathbf{k}\mathrm{r}\ \mathrm{in}\ \mathrm{s}
\end{array}$$

It should be noted, however, that for noepistemic descriptions $\approx^k$ is identical to strict entailment.

Having defined entailment of epistemic descriptions we will now define entailment based on weighted defaults. The syntactic format of weighted defaults is similar to the format used for rules.

Definition 11 *Let A be a* DL *alphabet. A* **weighted default** *δ has the form $c_1 \leadsto_n c_2$, where c_1 and c_2 are nonepistemic concepts, i.e. elements of $\mathcal{C}$, and n is a natural number. We call c_1 the premise of δ (written δ_p), c_2 its conclusion (written δ_c) and n its weight (written $w(\delta)$).*

In contrast to strict rules, weighted defaults allow for exceptions. The weight of a default is a measure of the likelihood of such an exception—the higher the weight, the less likely an exception.

Definition 12 *Let A be a* DL *alphabet, δ a weighted default, s a situation, and M a model over A. The* **epistemic exceptions** *to δ wrt s in M are defined as*

$$E^k_M(\delta, s) \quad \overset{\text{def}}{=} \quad \{o \in \mathcal{O} : [\![o]\!]^{\mathcal{I}} \in [\![\mathrm{k}\delta_p, s]\!]^{\mathcal{I}} \wedge [\![o]\!]^{\mathcal{I}} \notin [\![\mathrm{k}\delta_c, s]\!]^{\mathcal{I},\mathcal{W}}\}$$

In the following we will define a preferential semantics based on epistemic exceptions. It should be obvious how the corresponding definitions look for nonepistemic exceptions (see [Quantz, Suska 94] for details).

Given the exceptions to a default and its weight, we can assign negative scores to situations:

Definition 13 *Let A be a* DL *alphabet, M a model over A, s a situation, and Δ a finite set of weighted defaults. The* **negative score** *of M for s is defined as*

$$score^-_k(M, s) \quad \overset{\text{def}}{=} \quad \sum_{\delta\in\Delta}(|E^k_M(\delta, s)| \cdot w(\delta))$$

We can then order models wrt the negative score they obtain for situations.

Definition 14 *Let A be a* DL *alphabet and M, N two models over A.*
Σ-preference *is defined as*

$$\begin{array}{lll}
M \sqsubset^k_\Sigma N & \textit{iff} & 1.\forall s\ score^-_k(M) \geq score^-_k(N) \\
 & & 2.\exists s\ score^-_k(M) > score^-_k(N)
\end{array}$$

Note that this ordering on models is independent from any set of strict formulae Γ and only takes into account Δ and $\mathcal{O}$. Given some set Γ, we define $\sqsubset^k_\Sigma$-maximal models of Γ.

Definition 15 *Let $\mathcal{K} = \langle A, \Gamma\rangle$ be a knowledge base and M a model over A. M is a $\sqsubset^k_\Sigma$*-**maximal** *model of Γ iff M is a model of Γ and there is no model N of Γ such that $M \sqsubset^k_\Sigma N$.*

In the definition of the preferential entailment relation $\approx^k_\Sigma$ only $\sqsubset^k_\Sigma$-maximal models are taken into account:

Definition 16 *Let $\mathcal{K} = \langle A, \Gamma\rangle$ be a knowledge base, Δ a finite set of weighted defaults, $\sqsubset^k_\Sigma$ the corresponding ordering on models over A, and $\gamma \in \Phi_\alpha$. γ is* Σ-**entailed** *by Γ and Δ (written $\Gamma; \Delta \approx^k_\Sigma \gamma$) iff for all $\sqsubset^k_\Sigma$-maximal models M of Γ we have $M \models^k \gamma$.*

For examples of the behavior of $\approx^k_\Sigma$ and its formal properties see [Quantz 95].

C.3 A Proof Theory for Weighted Defaults

In this section we develop a proof theory for $\approx^k_\Sigma$ and show that it is decidable if the underlying DL is. The proof theory and its presentation is basically an adaptation of the ideas developed in [Quantz, Suska 94] to the epistemic case.

The main idea underlying the proof theory for PDDL are *default spaces*, which can be intuitively characterized as compact syntactic representations of models. In analogy to the techniques used in the preferential semantics we can score and order default spaces and thereby obtain maximal default spaces. The nonmonotonic entailment relation $\approx^k_\Sigma$ can then be reduced to strict entailment from maximal default spaces.

Definition 17 *Let $\langle A, \Gamma, \Delta\rangle$ be a default modeling. We define a set of* **default atoms** *A_Δ as follows:*

$$\begin{aligned}
A_p &\overset{\text{def}}{=} \{o :: \delta_p \textit{ in } s : o \in \mathcal{O}, \delta \in \Delta, s \in \mathcal{S}\}\\
A_c &\overset{\text{def}}{=} \{o :: \delta_c \textit{ in } s : o \in \mathcal{O}, \delta \in \Delta, s \in \mathcal{S}\}\\
A_\Delta &\overset{\text{def}}{=} A_p \cup A_c
\end{aligned}$$

Any subset of A_Δ is called a **default space** *S.*

In analogy to the scoring of models we assign each defaul space a negative score.

Definition 18 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and S a default space. The* **exceptions** *of S are defined as*

$$E_S(\delta, s) \overset{\text{def}}{=} \{o : o :: \delta_p \textit{ in } s \in S \ \wedge o :: \delta_c \textit{ in } s \notin S\}$$

The **negative score** *of S is defined as*

$$\begin{aligned} score_k^-(S, s) &\overset{\text{def}}{=} \sum_{\delta \in \Delta} (|E_S(\delta, s)| \cdot w(\delta)) \\ score_k^-(S) &\overset{\text{def}}{=} \sum_{S \in \mathcal{S}} score_k^-(S, s) \end{aligned}$$

Definition 19 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and S a default space. S is* Γ**-closed** *iff*

1. *$\Gamma \cup S$ is consistent;*
2. *$\forall \alpha \in A_\Delta \Gamma \cup S \models \alpha \Leftrightarrow \alpha \in S$.*

Obviously, a maximal default space is one which is Γ-closed and has minimal negative score.

Definition 20 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and S a default space. S is* $\sqsubset_\Sigma^k$**-maximal** *wrt Γ iff*

1. *S is Γ-closed;*
2. *there is no Γ-closed default space S' with $score_k^-(S') < score_k^-(S)$.*

We now define for each model the corresponding default space.

Definition 21 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and M a model of Γ.*

$$\begin{aligned} S_M \overset{\text{def}}{=} & \{o :: \delta_p \textit{ in } s : [\![o]\!]^{\mathcal{I}} \in [\![\mathbf{k}\delta_p, s]\!]^{\mathcal{I},\mathcal{W}}\} \cup \\ & \{o :: \delta_c \textit{ in } s : [\![o]\!]^{\mathcal{I}} \in [\![\mathbf{k}\delta_c, s]\!]^{\mathcal{I},\mathcal{W}}\} \end{aligned}$$

It is straightforward to prove that a model and its default space have the same negative score.

Lemma 1 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling, M a model of Γ, and S_M its default space.*

$$score_k^-(M) = score_k^-(S_M)$$

Proof:

$$\begin{aligned} score_k^-(M) &= \sum_{\mathrm{s}\in\mathcal{S}} score_k^-(M,\mathrm{s}) \\ &= \sum_{\mathrm{s}\in\mathcal{S}}\sum_{\delta\in\Delta}(|E_M^k(\delta,\mathrm{s})|\cdot w(\delta)) \\ &= \sum_{\mathrm{s}\in\mathcal{S}}\sum_{\delta\in\Delta}(|E_{S_M}(\delta,\mathrm{s})|\cdot w(\delta)) \\ &= \sum_{\mathrm{s}\in\mathcal{S}} score_k^-(S_M,\mathrm{s}) \\ &= score_k^-(S_M) \end{aligned}$$

The nontrivial step is the equivalence between $E_M^k(\delta,\mathrm{s})$ and $E_{S_M}(\delta,\mathrm{s})$:

$$\begin{aligned} E_M^k(\delta,\mathrm{s}) &= \{\mathrm{o}\in\mathcal{O} : [\![\mathrm{o}]\!]^{\mathcal{I}} \in [\![\mathrm{k}\delta_p,\mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \wedge [\![\mathrm{o}]\!]^{\mathcal{I}} \notin [\![\mathrm{k}\delta_c,\mathrm{s}]\!]^{\mathcal{I},\mathcal{W}}\} \\ &= \{\mathrm{o}\in\mathcal{O} : \mathrm{o} :: \delta_p \text{ in s} \in S_M \wedge \mathrm{o} :: \delta_c \text{ in s} \notin S_M\} \\ &= E_{S_M}(\delta,\mathrm{s}) \end{aligned}$$

□

For the non-epistemic semantics it was possible to establish a correspondence between maximal default spaces and maximal models such that all models of a maximal default space are maximal models. This is not possible for $\models\!\approx_\Sigma^k$, however. The reason is basically that a default space can be underdetermined wrt a default. Consider a default δ, a situation s, an object o, and the default spaces

$$\begin{aligned} S_1 &= \emptyset \\ S_2 &= \{\mathrm{o} :: \delta_p \text{ in s}\} \\ S_3 &= \{\mathrm{o} :: \delta_c \text{ in s}\} \\ S_4 &= \{\mathrm{o} :: \delta_p \text{ in s}, \mathrm{o} :: \delta_c \text{ in s}\} \end{aligned}$$

The first thing to note is that S_1, S_2, and S_4 have the same score and are all better than S_2. Secondly, all models of S_4 are models of S_1, S_2, and S_3, and all models of S_2 and S_3 are models of S_1. Thus if S_1 is a $\sqsubset_\Sigma^k$-maximal default space, it has models which are not necessarily maximal (namely the ones of S_2). We can prove, however, that any description satisfied in all maximal models of a maximal default space is also satisfied in all its non-maximal models.

To do so we need the following lemma:

Lemma 2 *Let A be a* DL *alphabet, o :: c in s $\in \Phi_\alpha$, and $M = \langle D_1, \mathcal{I}, \mathcal{W}_1\rangle$ $N = \langle D_2, \mathcal{J}, \mathcal{W}_2\rangle$ two models over A.*

$$\textit{If } M \sqsubset^k N \quad \textit{then} \quad N \models^k \textit{o :: c in s} \Rightarrow M \models^k \textit{o :: c in s}$$

Proof:

$$\begin{array}{rll} N \models^k \text{o :: c in s} & \text{iff} & \forall \mathcal{J}' \in \mathcal{W}_2 [\![\text{o}]\!]^{\mathcal{J}'} \in [\![\text{c, s}]\!]^{\mathcal{J}',\mathcal{W}_2} \\ & \Rightarrow & \forall \mathcal{J}' \in \mathcal{W}_1 [\![\text{o}]\!]^{\mathcal{J}'} \in [\![\text{c, s}]\!]^{\mathcal{J}',\mathcal{W}_1} \quad (\mathcal{W}_1 \subset \mathcal{W}_2) \\ & \text{iff} & M \models^k \text{o :: c in s} \end{array}$$

□

We can now prove that any description satisfied in all maximal models of a maximal default space is also satisfied in all its non-maximal models.

Lemma 3 *Let $\mathcal{K}_\Delta = \langle A, \Gamma, \Delta \rangle$ be a default modeling, $\gamma \in \Phi_\alpha$, S a $\sqsubset^k_\Sigma$-maximal default space wrt $\mathcal{K}_\Delta$, and M a model of S.*

1. *M is $\sqsubset^k_\Sigma$-maximal or*
2. *there is a $\sqsubset^k_\Sigma$-maximal model N of S with*
   $N \models^k \gamma \Rightarrow M \models^k \gamma$

Proof: Suppose M is not $\sqsubset^k_\Sigma$-maximal, i.e. we have a model M' with $M \sqsubset^k_\Sigma M'$. Clearly, there must be at least one o, δ, s, such that

$$\begin{array}{lll} M & \models^k & \text{o :: } \delta_p \textbf{ and not}(\delta_c) \text{ in s} \\ M' & \not\models^k & \text{o :: } \delta_p \textbf{ and not}(\delta_c) \text{ in s} \end{array}$$

Since M is a model of S we also know that o :: δ_c in s $\notin S$. Hence we can construct a model N from M by adding to $\mathcal{W}$ an interpretation function $\mathcal{J}$ with

$$[\![\text{o}]\!]^{\mathcal{J}} \in [\![\delta_p, \text{s}]\!]^{\mathcal{J},\mathcal{W}} \quad \wedge \quad [\![\text{o}]\!]^{\mathcal{J}} \in [\![\delta_c, \text{s}]\!]^{\mathcal{J},\mathcal{W}}$$

We can do so for all o, δ, s causing non-maximality of M and thereby obtain a $\sqsubset^k_\Sigma$-maximal model N. Furthermore we have $M \sqsubset^k N$ and thus $N \models^k \gamma \Rightarrow M \models^k \gamma$ (Lemma 2). □

Furthermore we can prove that the default space of a $\sqsubset^k_\Sigma$-maximal model is a $\sqsubset^k_\Sigma$-maximal default space:

Lemma 4 *Let M be a $\sqsubset^k_\Sigma$-maximal model. S_M is a $\sqsubset^k_\Sigma$-maximal default space.*

Proof: From Lemma 1 we know that M and S_M have the same negative score. Thus if M is $\sqsubset^k_\Sigma$-maximal, S_M must be $\sqsubset^k_\Sigma$-maximal too. □

With these lemmata we can finally prove that $\models\!\approx^k_\Sigma$ can be reduced to strict entailment from all maximal default spaces.

Proposition 1 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and $\gamma \in \Phi_\alpha$.*

$$\Gamma; \Delta \models\!\approx^k_\Sigma \gamma \quad \textit{iff} \quad \Gamma \cup S \models \gamma$$
$$\textit{in all } \sqsubset^k_\Sigma\textit{-maximal default spaces S over } \Gamma$$

Proof: (if) We have to prove that $\Gamma \cup S \models \gamma$ in all $\sqsubset^k_\Sigma$-maximal default spaces S over Γ implies $\Gamma; \Delta \approx^k_\Sigma \gamma$, i.e. that $M \models^k \gamma$ for all $\sqsubset^k_\Sigma$-maximal models of Γ.

Let M be such a $\sqsubset^k_\Sigma$-maximal model of Γ. From Lemma 4 we know that S_M is a $\sqsubset^k_\Sigma$-maximal default space and thus $\Gamma \cup S \models \gamma$, i.e. in all models N of $\Gamma \cup S$ we have $N \models^k \gamma$. M clearly is a model of $\Gamma \cup S$, hence $M \models^k \gamma$.

(only if) We have to prove that $\Gamma; \Delta \approx^k_\Sigma \gamma$ implies $\Gamma \cup S \models \gamma$ in all $\sqsubset^k_\Sigma$-maximal default spaces, i.e. all models M of $\Gamma \cup S$ must satisfy $M \models^k \gamma$.

Let S be such a $\sqsubset^k_\Sigma$-maximal default space and M any model of $\Gamma \cup S$. From Lemma 3 we know that M is either a $\sqsubset^k_\Sigma$-maximal model or we have a $\sqsubset^k_\Sigma$-maximal model N with $N \models^k \gamma \Rightarrow M \models^k \gamma$. Since we know from the premise that for all $\sqsubset^k_\Sigma$-maximal models L we have $L \models^k \gamma$, we get $M \models^k \gamma$ in both cases. □

A couple of remarks seem in order. First, this proposition shows that $\approx^k_\Sigma$ is decidable if the underlying DL is. Since $\mathcal{O}$, $\mathcal{S}$, and Δ are finite, there are only finitely many default spaces. To determine the maximal ones, we only have to check consistency in the underlying DL, more precisely, we have to enumerate all possible default spaces and pick the consistent sets with lowest score.

It is easy to see, however, that the number of default spaces is exponential, namely $2^{|\mathcal{O}| \cdot |\Delta| \cdot |\mathcal{S}|}$. Not all default spaces have to be considered by the algorithm, but even the number of maximal ones can, in the worst case, be exponential. (Consider a case when all sets of cardinality $> (|\mathcal{O}| \cdot |\Delta|)/2$ are inconsistent, all the others are consistent.) Since maximal default spaces cannot be subsets of each other, their number is bounded by $n!/(n$ DIV $2)!$, where $n = |\mathcal{O}| \cdot |\Delta|$.[1]

Finally the proof theory can be further simplified, namely by restricting it to *non-redundant* default spaces, thereby considerably reducing the average complexity of the algorithm presented in Section 4.2:

Definition 22 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and S a maximal default space. S is called* **non-redundant** *iff there is no maximal default space S' with $S' \subset S$.*

It is obvious that the formulae entailed by all non-redundant default spaces are the same as those entailed by all maximal default spaces:

Lemma 5 *Let $\langle A, \Gamma, \Delta \rangle$ be a default modeling and $\gamma \in \Phi_\alpha$.*

$$\begin{array}{ccc} \Gamma \cup S \models \gamma & \textit{iff} & \Gamma \cup S \models \gamma \\ \textit{for all maximal default spaces S} & & \textit{for all non-redundant default spaces S} \end{array}$$

[1] In [Suska 94] it is shown that even if the number of maximal default spaces is polynomially bounded, no algorithm can guarantee computation in polynomial time.

Proof: Obviously, $\Gamma \cup S_i \models \gamma$ for $i = 1, ..., n$ iff $\Gamma \cup (S_1 \cap ... \cap S_n) \models \gamma$. Redundant default spaces can thus be dropped from the conjunction since clearly $S_1 \subset S_2 \Rightarrow S_1 \cap S_2 = S_1$. $\square$

Note that this result means that a simple forward-chaining strategy is sufficient in the construction of maximal default spaces, i.e. if $\Gamma \not\models$ o :: δ_p in s we can ignore the default δ at object o in situation s (see Section 4.2 for details).

Appendix D

FLEX Manual

This chapter describes the syntax of FLEX in detail. In general, every entry in this manual consists of a heading line consisting of the construct's name[1] and a grouping keyword. In the case that several constructs have the same name (e.g. **domain**) we group them together. Entries consist of a short summary of the construct, its syntax in BNF, its formal semantics, a detailed description of the construct, some examples, specific differences to the BACK system and the construct's idiosyncrasies.

[1] For some operators like '..', ':<' etc. we use their Prolog functor as name. A functor consists of a predicate symbol and the number of it's arguments separated by '/'.

:</2 **Tell Expression**

Synopsis: Introduction of primitive term-names.

Syntax: ⟨*definition*⟩ ::= ⟨*term*-NAME⟩ :< ⟨*term*⟩[**with filter**=⟨*filter-list*⟩]

Semantics: $M \models^k \mathrm{t}_n :< \mathrm{t}$ iff $[\![\mathrm{t}_n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} \subseteq [\![\mathrm{t}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: The operator :<**/2** is used to introduce *primitive* terms into the knowledge base. For primitive terms only necessary but no sufficient conditions are given. Internally, an introduction of a primitive concept c :< **ctop** is transformed into the equivalent definition c := **prim**(c) **and ctop** (analogously for primitive roles). **prim**(c) is called a *primitive component*[2] and represents the information contained in c which is not completely specified by the user, i.e., which makes the term primitive.

Example:

publication	:<	**ctop**.
article	:<	publication.
publication_year	:<	**domain**(publication) **and range(number)** **and feature**.

See also: :=**/2**

[2] See [Nebel 90] for the technical details of introducing primitive components.

:=/2 Tell Expression

Synopsis: Introduction of defined term-names.

Syntax: ⟨*definition*⟩ ::= ⟨*term*-NAME⟩ := ⟨*term*⟩[**with filter**=⟨*filter-list*⟩]

Semantics: $M \models^k \mathrm{t}_n := \mathrm{t}$ iff $[\![\mathrm{t}_n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = [\![\mathrm{t}_n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: The operator :=**/2** is used to introduce *defined* terms into the knowledge base. A term-name is defined and can subsequently be used as an abbreviation of the term given as its definition. Unlike in the case of primitive terms the definition is taken to contain necessary and sufficient conditions, i.e., every instance satisfying the definition is taken to be an instance of the defined name.

Example:

scientific_book	:=	book **and** scientific_publication.
has_famous_author	:=	has_author **and range**(famous).

See also: :<**/2**

$=>$/2 **Tell Expression**

Synopsis: Specification of a rule.

Syntax: $\langle$*rule*$\rangle$ ::= $\langle$*concept*$\rangle$ $=>$ $\langle$*concept*$\rangle$

Semantics: $M \models^k \mathrm{c}_1 => \mathrm{c}_2$ iff $[\![\mathrm{kc}_1, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} \subseteq [\![\mathrm{kc}_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: A rule or constraint is specified: all instances of c_1 are constrained to be also instances of c_2. Both c_1 and c_2 can be complex concept terms. Rules between roles are not supported, however.

Example:

```
conference                 :<  ctop.
conference_publication     :<  publication.
ai_conference              :=  oneof([aaai,ijcai,ecai]).
at_conference              :<  domain(conference_publication)
                               and range(conference)
                               and feature.
the(at_conference,ai_conference)
                           =>  has_topic:artificial_intelligence
tractable_dl               ::  at_conference:ijcai.
tractable_dl               ?:  has_topic:artificial_intelligence.
```

Idiosyncrasy: Note that rules are not treated as material implications but only as forward chaining rules: if an object is known to be a c_1 it will be inferred that it is also a c_2. No contraposition or reasoning by case is performed.

Furthermore, rules are not applied to value restrictions. Thus, a rule $\mathrm{c}_1 => \mathrm{c}_2$ does not yield a subsumption between **all**(r,c_1) and **all**(r,c_2).

See also: $\sim n \sim>$**/2**

$\sim n \sim>$/2 **Tell Expression**

Synopsis: Specification of a weighted default.

Syntax: ⟨*default*⟩ ::= ⟨*concept*⟩~⟨INTEGER⟩~> ⟨*concept*⟩

Semantics: See Section C.2

Description: A weighted default is specified: an instance of c_1 is inferred to be an instance of c_2 if this is consistent with the knowledge base. The weight n specified the relevance of the default, the higher the weight the more relevant the default. In case of conflicting defaults, the weights are added to determine the globally preferred situtation.

To retrieve information based on defaults, the option **box=dbox** has to be used in **flexget**.

Example:

book **and the**(has_price,lt(10))	~20~>	buy_it:yes.
book_1	::	has_price:9.
book_1	?:	buy_it:yes **dbox**.

BACK: Defaults are not supported by the BACK system.

See also: =>/**2**, **?:/2**, **flexget**

::/2 Tell Expression

Synopsis: Enter new object descriptions into the knowledge base.

Syntax: $\langle$*description*$\rangle$ ::= $\langle$*object*-NAME$\rangle$ :: $\langle$*concept*$\rangle$[**in** $\langle$*sit-ref*$\rangle$]
| PROLOG-VAR :: $\langle$*concept*$\rangle$[**in** $\langle$*sit-ref*$\rangle$]

Semantics: $M \models^k \mathrm{o} :: \mathrm{c}\ \ \mathrm{in\ s\ iff}\ [\![\mathrm{o}]\!]^{\mathcal{J}} \in [\![\mathrm{c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: The operator **::/2** is used to enter new object descriptions into the knowledge base, and to create new objects. The left-hand side, $\langle$*object*-NAME$\rangle$ or PROLOG-VAR, determines whether the information of the right hand side, $\langle$*concept*$\rangle$, is asserted for an existing object or a new object. If $\langle$*object*-NAME$\rangle$ refers to an existing object then $\langle$*concept*$\rangle$ is added to the known object description. If $\langle$*object*-NAME$\rangle$ is a Prolog atom not associated with any object, a new object is created whose description is $\langle$*concept*$\rangle$, and whose name is $\langle$*object*-NAME$\rangle$. If PROLOG-VAR is an unbound Prolog variable, a new object is created whose description is $\langle$*concept*$\rangle$; the system generates an internal name, a *unique constant* of the form 'obj_n', and associates it with the object.

If any inconsistency occurs, the assertion is rejected, and **::/2** fails, e.g., if $\langle$*concept*$\rangle$ is incoherent or causes an inference that would introduce an inconsistency at another object.

If a situation is specified (**in** s), the description is valid in this situation and all situations extending it. If the specified situation is a variable, a new situation is generated. If no situation is specified, the description is added to the built-in situation **initial** which all other situations extend.

Example:

principia :: has_author:[russell,whitehead].
russell :: famous.
principia :: **all**(has_author,famous).
russian :: unknown_language in s1.

BACK: The BACK system supports plain object descriptions but no *situated* descriptions.

Idiosyncrasy: The user must neither create objects with the internally generated names ‘obj_N’, nor situations with the internally generated names ‘ext_sitN’.

See also: **?:/2**, **initial**

<<=/2 **Tell Expression**

Synopsis: Extends a situation.

Syntax:

$\langle$*tell-expression*$\rangle$ ::= $\langle$*sit-extension*$\rangle$

$\langle$*sit-extension*$\rangle$ ::= $\langle$*sit*-NAME$\rangle$<<= $\langle$*sit-ref*$\rangle$

Semantics: $[\![\mathrm{kc}, \mathrm{s}_1]\!]^{\mathcal{I},\mathcal{W}} \subseteq [\![\mathrm{kc}, \mathrm{s}_2]\!]^{\mathcal{I},\mathcal{W}} \forall \mathrm{c} \in \mathcal{C}$

Description: The situation of the right-hand side is an extension of the situation on the left-hand side. This means roughly speaking that all object descriptions holding in the situation on the left-hand side also hold in the situation on the right-hand side.

If the situation on the right-hand side is a variable, a situation name 'ext_sitN' is created by the system. Every situation extends the built-in situation **initial**.

Example:

russian :: unknown_language in s1.
s1 <<= s2.
russian ?: unknown_language in s2.

Idiosyncrasy: The user must not create situations with the internally generated names 'ext_sitN'.

See also: **::/2**, **?:/2**, **initial**

<>/2 Tell Expression

Synopsis: Marks the specified concepts as being mutually disjoint.

Syntax:

⟨*disjointness*⟩	::=	⟨*concept*-NAME⟩<> ⟨*concept*-NAME⟩
	|	<> '['⟨*concept*-NAME⟩{,⟨*concept*-NAME⟩}* ']'

Description: All the concepts listed in the argument list are marked as being mutually disjoint. This construct is provided as an abbreviation and is internally expanded into corresponding **negprim** terms.

Example:

```
scientific_publication              <> fiction.
scientific_publication              :< publication.
fiction                             :< publication.
scientific_publication and fiction ?< cbot.
```

Idiosyncrasy: Note that the <> operator can only be used for primitive concepts and that the <> statement must *precede* the definitions of the concepts which are marked as being disjoint.

See also: **disjoint**

?</2 Ask Expression

Synopsis: Subsumption test.

Syntax: $\langle$*ask-expression*$\rangle$ ::= $\langle$*term*$\rangle$?< $\langle$*term*$\rangle$

Semantics: $\Gamma \models \mathrm{t}_1 \ ? < \mathrm{t}_2$ iff $[\![\mathrm{t}_1, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \subseteq [\![\mathrm{t}_2, \mathrm{s}]\!]^{\mathcal{I},\mathcal{W}}$ in all models of Γ

Description: The operator performs a boolean test whether the $\langle$*term*$\rangle$ on the left-hand side is subsumed by the $\langle$*term*$\rangle$ on the right-hand side.

Example:

article	?<	publication.
european_country	?<	country.
scientific_book	:=	book **and** scientific_publication.
scientific_book	?<	book **and** scientific_publication.
book **and** scientific_publication	?<	scientific_book.

Idiosyncrasy: Note that the order of the arguments of **?**< and **subsumes** differs.

See also: **equivalent**, **subsumes**

?:/2 Ask Expression

Synopsis: Test whether an object instantiates a concept expression.

Syntax:

⟨*ask-expression*⟩	::=	⟨*object*-NAME⟩ ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩] [**dbox**]
	|	PROLOG-VAR ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩] [**dbox**]

Semantics: $\Gamma \models$ o ? : c in s iff $M \models^k$ o :: c in s in all models of Γ

Description: The operator performs a boolean test whether the object referred to by ⟨*object*-NAME⟩ instantiates the description given as ⟨*concept*⟩ in the specified situation. If no situation is specified, the test is performed in the built-in situation initial.

If **dbox** is specified, the test is performed by taking into account weighted defaults.

If a Prolog variable is used, all instances of the description are backtracked.

Example:

principia	?:	publication.
principia	?:	**some**(has_author,famous).
X	?:	**atleast**(2,has_author,famous).
		% binds X to principia
X	?:	has_written:principia.
		% binds X to russell; whitehead
book_4	?:	buy_it:yes **dbox**.

BACK: In the BACK system neither defaults nor situations are supported.

Idiosyncrasy: Variables on the left-hand side are only allowed if the state **class_objects** is set to **on**.

See also: $\sim n \sim>$**/2**, **::/2**, **flexstate**

$*$=/2 **Macro**

Synopsis: Operator for macro definition.

Syntax:

$\langle$*macro-definition*$\rangle$	::=	$\langle$*macro*$\rangle$ $*$= $\langle$*term*$\rangle$
$\langle$*macro*$\rangle$	::=	$\langle$*macro*-NAME$\rangle$[(VAR$\{$,VAR$\}^*$)]

See also: **flexmacro**

:/2 Concept Term

Synopsis: Specification of role-fillers.

Syntax:

⟨*concept*⟩	::=	⟨*role*⟩:⟨*value*⟩
	|	⟨*role*⟩:'['⟨*value*⟩{,⟨*value*⟩}*']'
	|	⟨*role*⟩:⟨*term*⟩
⟨*value*⟩	::=	⟨*object*-NAME⟩
	|	⟨*number*-INSTANCE⟩
	|	⟨*string*-INSTANCE⟩

Semantics: $[\![\mathrm{r{:}o}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{d : [\![\mathrm{o}]\!]^{\mathcal{J},\mathcal{W}} \in [\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}(d)\}$

Description: An object is specified as a role-filler for a role. This operator allows the use of constants in concept definitions. 'r:[o_1,o_2]' is equivalent to 'r:o_1 **and** r:o_2'.

For term-valued features the filler is a concept or a role.

Example:

chinese_room	::	has_author:searle.
principia	::	has_author:[russell,whitehead].
flex	::	has_topic:description_logics.

Idiosyncrasy: Descriptions of role-fillers have no impact on concept subsumption.

See also: **fillers**

../2 Number Term

Synopsis: Constructs a closed numerical interval from a lower and an upper bound.

Syntax: ⟨*number-range*⟩ ::= ⟨*lower-limit*⟩
| ⟨*upper-limit*⟩
| ⟨*lower-limit*⟩ .. ⟨*upper-limit*⟩

Semantics: $[\![n_1..n_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{m \in N_0 : n_1 \leq m \leq n_2\}$

Description: This operator is used to represent the closed numerical interval between the given lower and upper limit, where lower and upper limit are numbers. An interval where the lower limit is equal to the upper limit contains just a single value.

Example: sixties_publication := publication **and**
the(publication_year,1960..1969).

Idiosyncrasy: Note that the range N..N is equivalent to the number N.

See also: **le, lt**, **ge, gt**

all — Concept Term/Method

Synopsis: Specifies and retrieves value restrictions.

Syntax:

⟨*concept*⟩	::=	**all**(⟨*role*⟩,⟨*concept*⟩)
⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**all**(⟨*role*⟩[,⟨*sit*-NAME⟩])

Semantics: $[\![\forall \text{r:c}, \text{s}]\!]^{\mathcal{J},\mathcal{W}} = \{d : [\![\text{r}, \text{s}]\!]^{\mathcal{J},\mathcal{W}}(d) \subseteq [\![\text{c}, \text{s}]\!]^{\mathcal{J},\mathcal{W}}\}$

Description: The concept constructor **all** means that all fillers for role r must be of type c. Note that this restricts only the fillers locally at a concept. To restrict the fillers of a role globally, the **range** operator must be used.

The method **all** retrieves the value restriction of a concept or an objects for a role. For objects the situation has to be specified.The option **box** can be used to take rules and defaults into account during retrieval (see **flexget** for details).

Example: principia :: **all**(has_author,famous).
flexget(arithmetik,**all**(has_author),Result).

```
% binds Result to famous
```

Idiosyncrasy: This construct does not imply the existence of any role-filler. Objects having no role-filler for role r (i.e. instances of ‘**atmost**(0,r)’), are trivially instances of **all**(r,c) for arbitrary c.

See also: **flexget**, **no**, **some**, **range**

and — Concept/Role Term

Synopsis: Conjunction of concepts and roles.

Syntax:

⟨*concept*⟩	::=	⟨*concept*⟩ **and** ⟨*concept*⟩
⟨*role*⟩	::=	⟨*role*⟩ **and** ⟨*role*⟩

Semantics: $[\![t_1 \sqcap t_2, s]\!]^{\mathcal{J},\mathcal{W}} = [\![t_1, s]\!]^{\mathcal{J},\mathcal{W}} \cap [\![t_2, s]\!]^{\mathcal{J},\mathcal{W}}$

Description: The operator **and** is the basic construct for combining terms, and can be used both for concepts and for roles. The resulting term represents the conjunction of both terms.

Example:

publication_year	:<	**domain**(publication) **and range(number) and feature**.
sixties_publication	:=	publication **and the**(publication_year,1960..1969).
origin	::	book **and** scientific_publication.

See also: **or**, **not**

atleast — Concept Term/Method

Synopsis: Specifies and retrieves (qualifying) minimum restriction of roles.

Syntax:

⟨*concept*⟩	::=	**atleast**(⟨INTEGER⟩,⟨*role*⟩[,⟨*concept*⟩])
⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**atleast**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])

Semantics: $[\![\geq n\ \mathrm{r{:}c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{d : |[\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}(d) \cap [\![\mathrm{c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}| \geq n\}$

Description: The plain minimum restriction **atleast**(n,r) means that there are at least n role-fillers for role r; the qualifying minimum restriction **atleast**(n,r,c) means that there are at least n role-fillers of type c for role r.

The method **atleast** retrieves the (qualified) minimum restriction for the specified role at a concept or an object. For objects the situation has to be specified. The option **box** can be used to take rules and defaults into account during retrieval (see **flexget** for details).

Example:

atleast(2,has_author,famous)	?<	**atleast**(2,has_author **and range**(famous)).
group_publication	:=	**atleast**(3,has_author).
X	?:	**atleast**(2,has_author,famous). % binds X to principia

flexget(group_publication,**atleast**(has_author),Result).
```
% binds Result to 3
```
flexget(principia,**atleast**(has_author,famous),Result).
```
% binds Result to 2
```

See also: **atmost**, **exactly**, **flexget**, **some**, **no**, **the**

atmost — Concept Term/Method

Synopsis: Specifies and retrieves (qualifying) maximum restriction of roles.

Syntax:

⟨*concept*⟩	::=	**atmost**(⟨INTEGER⟩,⟨*role*⟩)
	|	**atmost**(⟨INTEGER⟩,⟨*role*⟩,⟨*concept*⟩)
⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**atmost**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])

Semantics: $[\![\leq n \text{ r:c}, \text{s}]\!]^{\mathcal{J},\mathcal{W}} = \{d : |[\![\text{r}, \text{s}]\!]^{\mathcal{J},\mathcal{W}}(d) \cap [\![\text{c}, \text{s}]\!]^{\mathcal{J},\mathcal{W}}| \leq n\}$

Description: The plain maximum restriction **atmost**(n,r) means that there are at most n role-fillers for role r; the qualifying maximum restriction **atmost**(n,r,c) means that there are at most n role-fillers of type c for role r.

The method **atmost** retrieves the (qualified) maximum restriction for the specified role at a concept or an object. For objects the situation has to be specified. The option **box** can be used to take rules and defaults into account during retrieval (see **flexget** for details).

Example: individual_publication := **atmost**(1,has_author).
flexget(individual_publication,**atmost**(has_author),Result).
```
% binds Result to 1
```
flexget(arithmetik,**atmost**(has_author,famous),Result).
```
% binds Result to 1
```

See also: **atleast**, **exactly**, **some**, **no**, **the**, **flexget**

cbot — Concept Term

Synopsis: The incoherent concept.

Syntax: $\langle$*concept*$\rangle$ ::= **cbot**

Semantics: $[\![\bot]\!]^{\mathcal{I},\mathcal{W}} = \emptyset$

Description: The incoherent concept which has no instances at all. It can be used to check the incoherence of concepts, e.g., c ?< **cbot**, or to check whether value restrictions are incoherent, e.g., c ?< **all**(r,**cbot**). It is also useful for specifying non-primitive disjointness via rules, e.g., **atleast**(2,r) **and atmost**(3,s) => **cbot**.

Example:

scientific_publication	<>	fiction.
scientific_publication	:<	publication.
fiction	:<	publication.
scientific_publication **and** fiction	?<	**cbot**.

BACK: The incoherent concept is called **nothing** in BACK.

See also: **ctop**, **rbot**, **disjoint**, **incoherent**

comp — Role Term

Synopsis: Infix operator for composition of two roles.

Syntax: $\langle role \rangle$ ::= $\langle role \rangle$ **comp** $\langle role \rangle$

Semantics: $[\![\mathrm{r}_1.\mathrm{r}_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = [\![\mathrm{r}_1, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} \circ [\![\mathrm{r}_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: The role operator **comp** produces the composition of two roles.

Example:

```
from_institution :=has_author comp at_institution.
from_country     :=from_institution comp in_country.
```

Idiosyncrasy: FLEX computes fillers for chains only for “active role chains”. For roles which are defined as compositions, the systems asserts an active role chain itself, e.g. in the above example ‘active_role_chain(has_author,at_institution).’ is asserted. If you want fillers to be inferred for compositions which are not defined roles, you have to assert these predicates yourself.

See also: **inv**, **domain**, **range**

concepts Method

Synopsis: Retrieves all concepts instantiated by an object.

Description: The method **concept** retrieves all concepts an object instantiates. If **box=dbox** these concepts are determined by taking into account weighted defaults; otherwise only definitions, descriptions and rules are taken into account.

If a **filter** is specified, only concepts satisfying the filter are considered.

Example: flexget(principia,concepts(initial),Result).

```
% binds Result to [ctop,publication,book]
```

See also: **instances**, **msc**

ctop — Concept Term

Synopsis:	Built-in topmost concept.
Syntax:	$\langle$*concept*$\rangle$ ::= **ctop**
Semantics:	$[\![\top]\!]^{\mathcal{I},\mathcal{W}} = D$
Description:	**ctop** is the topmost concept and can be used to build primitive concept hierarchies.
Example:	publication $:<$ **ctop**.
BACK:	The top-most concept in BACK is called **anything**.
See also:	**cbot**, **rtop**

dir_subs — Method

Synopsis: Retrieves the direct subsumees of a term.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**dir_subs**

Description: The method **dir_subs** is applicable to concepts and roles and retrieves their direct subsumees. It can thus be used to traverse the term hierarchies, e.g. to build a graphical representation. Meaningful options are **box** and **filter**.

The option **box** can be used to take into account rules and weighted defaults. If no box is specified the subsumption hierarchy stemming from definitions is taken; if **box=ibox**, the subsumption hierarchy also takes into account rules; if **box=dbox** even defaults are taken into account.

If the option **filter** is used, only terms satisfying the filter are taken into account in the subsumption hierarchy.

Example: **flexget**(book,**dir_subs**,Result).
% binds Result to [novel,scientific_book]

See also: **flexget**, **dir_supers**, **subs**, **filter**

dir_supers Method

Synopsis: Retrieves the direct subsumers of a term.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**dir_supers**

Description: The method **dir_supers** is applicable to concepts and roles and retrieves their direct subsumers. It can thus be used to traverse the term hierarchies, e.g. to build a graphical representation. Meaningful options are **box** and **filter**.

The option **box** can be used to take into account rules and weighted defaults. If no box is specified the subsumption hierarchy stemming from definitions is taken; if **box=ibox**, the subsumption hierarchy also takes into account rules; if **box=dbox** even defaults are taken into account.

If the option **filter** is used, only terms satisfying the filter are taken into account in the subsumption hierarchy.

Example: **flexget**(scientific_book,**dir_supers**,Result).

```
% binds Result to
[scientific_publication,book]
```

See also: **flexget**, **dir_subs**, **supers**, **filter**

disjoint — Ask Expression

Synopsis: Tests the disjointness of terms.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
⟨*ask-expression*⟩	::=	**disjoint**(⟨*term*⟩,⟨*term*⟩)

Description: **disjoint** performs a boolean test to determine whether the two terms given as arguments are disjoint, i.e., it conjoins both terms and determines whether the conjoined definition is subsumed by **cbot** or **rbot**.

If no box is specified, the disjointness test is performed on the basis of the term definitions; if **box=ibox** rules are also taken into account; if **box=dbox** rules and weighted defaults are taken into acount.

Example: **flexask**(**disjoint**(scientific_publication,fiction))

See also: **?</2**, **incoherent**, **cbot**, **rbot**, **flexask**

domain — Role Term/Method

Synopsis: Restricts the domain of a role to the specified concept.
Retrieves the domain of a role.

Syntax:

⟨*role*⟩	::=	**domain**(⟨*concept*⟩)
⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	domain

Semantics: $[\![\mathrm{c}|\mathrm{r},\mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} = [\![\mathrm{r},\mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \cap ([\![\mathrm{c},\mathrm{s}]\!]^{\mathcal{I},\mathcal{W}} \times D)$

Description: The role operator **domain** restricts the domain of a role, i.e. the first argument of a role instance has to be an instance of the specified concept. The domain has to be a “real” *concept*, instances of **number** and **string** are not allowed to have role-fillers.

The method **domain** retrieves the domain of a role.

Example: publication_year :< **domain**(publication) **and range(number)**
and feature.
flexget(has_author,**domain**,Result).
`% binds Result to publication`

Idiosyncrasy: Defined role introductions, containing only a domain restriction, i.e., r := **domain**(c), are forbidden.

See also: **range**, **flexget**

equivalent — Ask Expression

Synopsis: Tests the equivalence of terms.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
⟨*ask-expression*⟩	::=	**equivalent**(⟨*term*⟩,⟨*term*⟩)

Description: **equivalent** performs a boolean test to determine whether the two terms given as arguments are equivalent, i.e., it tests whether each term is subsumed by the other.

If no box is specified, the test is performed on the basis of the term definitions; if **box=ibox** rules are also taken into account; if **box=dbox** rules and weighted defaults are taken into acount.

Example: **flexask**(**equivalent**(scientific_book,book **and** scientific_publication))

See also: **?</2**, **subsumes**

equivalents Method

Synopsis: Retrieves equivalent terms.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**equivalents**

Description: The method **equivalents** is applicable to concepts and roles and retrieves equivalent concepts or roles. Meaningful options are **box** and **filter**.

The option **box** can be used to take into account rules and weighted defaults. If no box is specified equivalence is tested only onb the basis of definitions; if **box=ibox**, the rules are taken into account; if **box=dbox** defaults are also taken into account.

If the option **filter** is used, only terms satisfying the filter are taken into account.

See also: **flexget**

exactly — Concept Term

Synopsis: (Qualifying) minimum and maximum restriction.

Syntax:

$\langle$*concept*$\rangle$ ::= **exactly**($\langle$INTEGER$\rangle$,$\langle$*role*$\rangle$)
| **exactly**($\langle$INTEGER$\rangle$,$\langle$*role*$\rangle$,$\langle$*conceptual-type*$\rangle$)

Description: There are exactly n role-fillers at role r, resp. there are exactly n role-fillers of type c

Example: arithmetik :: **exactly**(1,has_author).

Idiosyncrasy: Internally, the restrictions '**exactly**(n,r)' and '**exactly**(n,r,c)' are transformed into '**atleast**(n,r) **and atmost**(n,r)' and '**atleast**(n,r,c) **and atmost**(n,r,c)', respectively.

See also: **atleast**, **atmost**

feature — Role Term

Synopsis: Specifies a role as being a feature, i.e. functional.

Syntax: $\langle$*role*$\rangle$ ::= **feature**

Semantics: $\langle d_1, d_2\rangle \in [\![\mathrm{f}_p, \mathrm{s}]\!]^{\mathcal{I}}, \langle d_1, d_3\rangle \in [\![\mathrm{f}_p, \mathrm{s}]\!]^{\mathcal{I}} \Rightarrow d_2 = d_3$

Description: Features are roles which are functional, i.e. no object can have more than one filler for a feature. Note that features are partial functions, i.e. there may be objects having no filler at all for a feature.

Example: publication_year :< **domain**(publication) **and range(number) and feature**.

See also: **:/2**

fillers **Method**

Synopsis: Retrieves the fillers at a role.

Syntax:

$\langle$*interaction*$\rangle$ ::= **flexget**($\langle$*entity*$\rangle$,$\langle$*method*$\rangle$,[$\langle$*options*$\rangle$,]VAR)
$\langle$*method*$\rangle$ ::= **fillers**($\langle$*role*$\rangle$[,$\langle$*sit*-NAME$\rangle$])

Description: To retrieve the role-fillers of an object or a concept, the method **fillers** can be used. Depending on the specified option for **box**, rules (**box=ibox**) and defaults (**box=dbox**) are taken into account to determine the fillers.

Example: **flexget**(principia,**fillers**(has_author,**initial**),Result).

```
% binds Result to [russell,whitehead]
```

See also: **:/2**, **flexget**

filter Method

Synopsis: Retrieves the filters satisfied by a term.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**filter**

Description: Returns the filters associated with a concept or a role. Note that filters allow a limited form of representing second-order information, i.e. properties of concepts or roles. Filters apply only to the term at which they have been specified and are not inherited.

The following filters are built-in:

dbox_lhs is satisfied by concepts occurring on the left-hand sides of defaults.

dbox_rhs is satisfied by concepts occurring on the right-hand sides of defaults.

ibox_lhs is satisfied by concepts occurring on the left-hand sides of rules.

ibox_rhs is satisfied by concepts occurring on the right-hand sides of rules.

user_defined is satisfied by concepts or roles which are defined by the user.

See also: **flexget**

flexask — Interaction

Synopsis: Performs a boolean test as specified by the argument.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩)[,box=⟨*box*⟩]
⟨*ask-expression*⟩	::=	⟨*term*⟩ ?< ⟨*term*⟩
	|	⟨*object*-NAME⟩ ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩]
	|	PROLOG-VAR ?: ⟨*concept*⟩[**in** ⟨*sit*-NAME⟩]
	|	**disjoint**(⟨*term*⟩,⟨*term*⟩)
	|	**subsumes**(⟨*term*⟩ ⟨*term*⟩)
	|	**equivalent**(⟨*term*⟩,⟨*term*⟩)
	|	**incoherent**(⟨*term*⟩)
	|	**satisfies**(⟨*term*-NAME⟩,⟨*filter-list*⟩)

Description: This operator is used to perform boolean tests regarding subsumption, disjointness, equivalence, incoherence, or satisfaction of filters.

If no box is specified the tests are performed on the basis of definitions only; if **box=rules** rules are taken into account as well; if **box=defaults** rules and defaults are taken into account.

Example: **flexask**(**disjoint**(scientific_publication,fiction))

Idiosyncrasy: The **flexask** operator can be omitted for expressions containing the operators **?</2** or **?:/2**, since they uniquely identify an expression as a **flexask**.

See also: **?</2**, **?:/2**, **disjoint**, **equivalent**, **satisfies**, **incoherent**, **subsumes**

flexdump Interaction

Synopsis: Dumps the internal representation of the current knowledge base.

Syntax: ⟨*interaction*⟩ ::= **flexdump**(⟨*file*-NAME⟩[,Comment])

Description: **flexdump** dumps the the contents of the knowledge base into the file specified in ⟨*file*-NAME⟩. A dumped knowledge base can be loaded with **flexload**.

A comment can be specified as second argument which will be output when the knwoledge base is loaded.

Example: **flexdump**('MyFavoriteFilename').
flexdump('MyFavoriteFilename','Version 2.3')

BACK: In the BACK system **flexdump** could be called without an argument, in which case the knowledge base was dumped to the standard output.

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

See also: **flexload**

flexget Interaction

Synopsis: Retrieves information about entities.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**all**(⟨*role*⟩[,⟨*sit*-NAME⟩])
	|	**atleast**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])
	|	**atmost**(⟨*role*⟩[,⟨*concept*⟩] [,⟨*sit*-NAME⟩])
	|	**concepts**(⟨*sit*-NAME⟩)
	|	**dir_subs**
	|	**dir_supers**
	|	**domain**
	|	**equivalents**
	|	**fillers**(⟨*role*⟩[,⟨*sit*-NAME⟩])
	|	**filter**
	|	**help**
	|	**instances**(⟨*sit*-NAME⟩)
	|	**msc**(⟨*sit*-NAME⟩)
	|	**range**
	|	**subs**
	|	**supers**
	|	**tvf_filler**(⟨*role*⟩[,⟨*sit*-NAME⟩])
⟨*options*⟩	::=	box=⟨*box*⟩
	|	filter=⟨*filter-list*⟩

Description: Whereas **flexask** only performs boolean tests, **flexget** can be used to retrieve specific information about an entity. The retrieved information is bound to a Prolog variable.

The option **box** specifies on which basis the information is retrieved. Note that this does depend both on the option and on the fact whether the entity is a concept or an object:

*Entity**Box Option*	default	**box=rules**	**box=defaults**
concepts	definitions	rules definitions	rules definitions defaults
objects	definitions descriptions rules	definitions descriptions rules	definitions descriptions rules, defaults

It is thus not possible to retrieve information about objects without taking into account rules.

The option **filter** can be used to restrict the concept or role names which are returned to names satisfying a specific filter. This can be useful if a component is not interested in internally generated names, or is only interested in a subset of the hierarchy.

Example: **flexget**(principia,**fillers**(has_author,**initial**),Result).

```
% binds Result to [russell,whitehead]
```

See also: **flexask**

flexinit Interaction

Synopsis: Initializes the FLEX system.

Syntax: ⟨*interaction*⟩ ::= **flexinit**

Description: **flexinit** initializes the FLEX system completely, i.e. all internal data structures are initialized, the state variables are set back to their values at start-up time (see below), and all previously entered definitions, rules, defaults , descriptions, and macros are removed.

flexinit is usually the first tell in a file read in by FLEX.

State	*Settings*	*Setting after* FLEXINIT
class_objects	on, off	on
defspace_dbox	best,all	best
eval_dbox	local,global	local
introduction	forward,noforward	noforward
syntax_check	on,off	on
verbosity	silent,error,warning,info,trace	info

See also: **flexstate**, **flexread**

flexload — Interaction

Synopsis: Loads a dumped internal representation from a file.

Syntax: ⟨*interaction*⟩ ::= **flexload**(⟨*file*-NAME⟩)

Description: **flexload** loads a previously dumped knowledge base from file ⟨*file*-NAME⟩ back into the FLEX system, so that the state of the FLEX system is restored to the state of FLEX before the knowledge base was dumped.

Example: **flexload**('MyFavoriteFilename').

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

See also: **flexdump**, **flexread**

flexmacro **Interaction**

Synopsis: Definition of a macro.

Syntax: $\langle$*interaction*$\rangle$ ::= **flexmacro**($\langle$*macro-definition*$\rangle$)

Description: The macro-facility can be used to define new term-forming operators or to rename existing term-forming operators. Note that the Prolog-variables occurring in the term on the right-hand side must all be bound by arguments on the left-hand side of the macro.

Example: **flexmacro**(all1(R,C) $*=$ **all**(R,C) **and atleast**(1,R)).
flexmacro(min(N,R) $*=$ **atleast**(N,R)).

Idiosyncrasy: The macro-facility is restricted to macros for terms. Whole interaction sequences with FLEX can be easily defined as Prolog-predicates:

my_init :- **flexinit**,
flexstate(verbosity=warnings).

flexread — Interaction

Synopsis: Reads FLEX commands from a file.

Syntax: ⟨*interaction*⟩ ::= **flexread**(⟨*file*-NAME⟩)

Description: Reads from the specified file interaction operations for building up and accessing a FLEX knowledge base.

The file read by **flexread** may contain further interaction operations. Thus, it is possible to issue **flexinit**, **flexread**, **flexload**, and **flexdump** operations from the file read.

For assuring that a file is read into an initialized empty FLEX system the file should contain as first statement a **flexinit**. For assuring that FLEX understands some predefined macros a further statement can be explicitly issued to load them: **flexread**(<macro_file_name>).

Depending on the verbosity setting, messages will be produced and written to the current standard output.

Example: **flexread**('DOCU/examples.model').

Idiosyncrasy: The form of the file name depends on your local site, but should be quoted according to the Prolog convention if it contains special characters.

Although a file read by **flexread** may contain arbitrary calls to Prolog, it should be noted that this is an additional feature of the Prolog implementation of FLEX, which might not be supported in other implementations.

See also: **flexload**, **flexdump**, **flexstate**

flexstate **Interaction**

Synopsis: Displays, modifies or retrieves the values of global state variables.

Syntax:

⟨*interaction*⟩	::=	**flexstate**[(⟨*state*⟩)]
⟨*state*⟩	::=	**verbosity** = **silent**
	|	**verbosity** = **error**
	|	**verbosity** = **warning**
	|	**verbosity** = **info**
	|	**verbosity** = **trace**
	|	**introduction** = **forward**
	|	**introduction** = **noforward**
	|	**class_objects** = **on**
	|	**class_objects** = **off**
	|	**syntax_check** = **on**
	|	**syntax_check** = **off**

Description: **flexstate** can be used to set FLEX states which determine the behavior of the system:

verbosity determines which types of output messages are produced.

silent no output is produced

errors only errors are reported

warnings errors and warnings are issued

info additional information is reported

trace produces an exhaustive trace of what happens in BACK.

introduction determines whether undefined names are introduced automatically.

noforward Forward introduction of names is not performed. Thus, names have to be defined before they are used.

forward If a name is used without being previously defined, it will be introduced automatically as a primitive term.

Note that while forward introduction is useful for small test cases, it may be problematic for modeling large domains—

spelling errors will be difficult to detect, since misspelled terms are automatically introduced as new terms.

class_objects determines whether objects are classified wrt the conceptual hierarchy or not. Note that the method **instances** can only be used if **class_objects** is set to **on**.

syntax_check determines whether the tells contained in a file read in by **flexread** are first checked for syntactic correctness.

Instead of the full name of state variables and their values unambiguous abbreviations can be used.

Example: **flexstate**(**verbosity** =**errors**).
flexstate(ve = err).

Idiosyncrasy: Note that the **flexinit** resets all states to their default values. To keep your preferred sates you should write your own initialization routine as a Prolog predicate, e.g.:

my_init :- **flexinit**,
flexstate(verbosity=warnings).

See also: **flexinit**

flextell **Interaction**

Synopsis: Tells the FLEX system the information conveyed in the argument.

Syntax:

⟨*interaction*⟩	::=	**flextell**(⟨*tell-expression*⟩)
⟨*tell-expression*⟩	::=	⟨*definition*⟩
	|	⟨*rule*⟩
	|	⟨*description*⟩
	|	⟨*default*⟩
	|	⟨*macro-definition*⟩
	|	⟨*disjointness*⟩
	|	⟨*sit-extension*⟩

Description: This operator is used to assert new information in the form of tell-expressions.

Example: **flextell**(publication :¡ ctop).

Idiosyncrasy: You may drop the **flextell** for tell-expressions consisting of the operators **:=/2**, **:</2**, **=>/2**, $\sim n \sim$**>/2** or **::/2**, since they uniquely identify an expression as a **flextell**.

See also: **flexask**, **flexget**

ge, gt — Number Term

Synopsis: Constructs a numerical interval with infinite upper bound.

Syntax: $\langle$*lower-limit*$\rangle$ ::= **ge**($\langle$*number*-INSTANCE$\rangle$)
| **gt**($\langle$*number*-INSTANCE$\rangle$)

Semantics:

$[\![> n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{m \in N_0 : m > n\}$
$[\![\geq n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{m \in N_0 : m \geq n\}$

Description: These operators construct from a given number an interval with an infinite upper bound and either closed lower bound (in case of **ge**) or open lower bound (in case of **gt**).

Example:
human :¡ **ctop**.
has_age :¡ **range**(**number**).
adult :=human **and** has_age(**ge**(18))

See also: **../2**, **le, lt**, **number**

help Method

Synopsis: Retrieves the methods available for an entity.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**help**

Description: The method **help** retrieves all methods available for an entity. No options are meaningful for this method.

See also: **flexget**

incoherent — Ask Expression

Synopsis: Tests whether a term is incoherent.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
⟨*ask-expression*⟩	::=	**incoherent**(⟨*term*⟩)

Description: **incoherent** tests whether the term given as argument is incoherent, i.e., whether it is subsumed by **cbot** or **rbot**.

If no box is specified, the test is performed on the basis of the term definitions; if **box=ibox** rules are also taken into account; if **box=dbox** rules and weighted defaults are taken into acount.

Example: **flexask**(**incoherent**(fiction **and** scientific_publication))

See also: **?</2**, **disjoint**, **cbot**, **rbot**

initial — Situation

Synopsis: Built-in situation.

Description: The built-in situation **initial** is extended by all other situations. If an object description does not contain a situation it is take to be a description of **initial**. Similarly, an instanceship query which does not contain a situation is evaluated wrt ‘initial’.

Example:

```
flexget(principia,msc(initial),Result).
% binds Result to [book]
```

Idiosyncrasy: In tells and queries you do not have to specify the built-in situation **initial**. In **flexget** you have to specify a situation, however, to disambiguate between methods for concepts and methods for objects.

See also: **::/2**, **?:/2**, **flexget**

inv — Role Term

Synopsis: Construction of inverse roles.

Syntax: $\langle$*role*$\rangle$ ::= **inv**($\langle$*role*$\rangle$)

Semantics: $[\![\mathrm{r}^{-}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{\langle d_1, d_2\rangle : \langle d_2, d_1\rangle \in [\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}\}$

Description: **inv** is a role operator for defining the inverse of a role. The argument of **inv** may be a role name or an arbitrary role term.

Example:

```
has_written := inv(has_author).
X           ?: has_written:principia.
               % binds X to russell;whitehead
```

Idiosyncrasy: Note that you cannot invert a role if its range is of type **number** or **string**.

See also: **comp**, **domain**, **range**

instances Method

Synopsis: Retrieves all instances of a concept

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**instances**(⟨*sit*-NAME⟩)

Description: All instances of a concept in the specified situation are retrieved.

Example: flexget(article,instances(initial),Result).

```
% binds Result to
[chinese_room,multi_pub,tractable_dl,obj_2]
```

See also: **?:/2**, **concepts**, **msc**, **flexget**

le, lt — Number Term

Synopsis: Constructs a numerical interval with infinite lower bound.

Syntax: ⟨*upper-limit*⟩ ::= **le**(⟨*number*-INSTANCE⟩) | **lt**(⟨*number*-INSTANCE⟩)

Semantics:
$[\![< n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{m \in N_0 : m < n\}$
$[\![\leq n, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{m \in N_0 : m \leq n\}$

Description: These operators construct from a given number an interval with infinite lower bound and an either closed upper bound (in case of **le**) or an open upper bound (in case of **lt**).

Example: book **and the**(has_price,**lt**(10)) ∼20∼> buy_it:yes.

See also: **../2**, **ge, gt**, **number**

msc Method

Synopsis: Retrieves the most specific concepts an object instantiates.

Syntax:

$\langle$*interaction*$\rangle$ ::= **flexget**($\langle$*entity*$\rangle$,$\langle$*method*$\rangle$,[$\langle$*options*$\rangle$,]VAR)
$\langle$*method*$\rangle$ ::= **msc**($\langle$*sit*-NAME$\rangle$)

Description: The method **msc** retrieves the most specific concepts an object instantiates. If **box=dbox** these concepts are determined by taking into account weighted defaults; otherwise only definitions, descriptions and rules are taken into account.

If a **filter** is specified, only concepts satisfying the filter are considered.

Example: **flexget**(principia,**msc(initial)**,Result).

```
% binds Result to [book]
```

See also: **flexget**, **concepts**, **instances**

no — Concept Term

Synopsis:	(Qualifying) nonexistence restriction.
Syntax:	$\langle$*concept*$\rangle$ ::= **no**($\langle$*role*$\rangle$,$\langle$*conceptual-type*$\rangle$) $\mid$ **no**($\langle$*role*$\rangle$)
Description:	There are no role-fillers at role r, resp. there are no role-fillers of type c at role r.
Idiosyncrasy:	Internally, the restrictions '**no**(r)' and '**no**(r,c)' are transformed into '**atmost**(0,r)' and into '**atmost**(0,r,c)', respectively.
See also:	**all**, **atleast**, **atmost**, **exactly**,**some**, **the**

not — Concept/Role Term

Synopsis: Negation of concepts and roles

Syntax:

$\langle \textit{concept} \rangle$ ::= **not**($\langle \textit{concept} \rangle$)

$\langle \textit{role} \rangle$ ::= **not**($\langle \textit{role} \rangle$)

Semantics:

$[\![\neg\mathrm{c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = D \setminus [\![\mathrm{c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

$[\![\neg\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = (D \times D) \setminus [\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$

Description: The **not** operator can be used to negate concepts and roles. The use of the **not** operator will usually lead to disjunctive normal-forms, which in turn may lead to inefficient performance. To define concepts as being disjoint it may be more efficient to use $<>$.

See also: $<>$, **and**, **or**

number — Concept Term

Synopsis: Built-in topmost number.

Syntax: $\langle$*number*$\rangle$ ::= **number**

Semantics: $[\![\top_{\mathrm{n}}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = N_0$

Description: **number** is the topmost number.

Example: publication_year :< **domain**(publication) **and range(number)**
and feature.

See also: **../2**, **ge,gt**, **le,lt**

oneof — Concept Term

Synopsis: Extensional concept definition.

Syntax: ⟨*concept*⟩ ::= **oneof**('['⟨*object*-NAME⟩){,⟨*object*-NAME⟩}*']'

Semantics: $[\![\{\mathrm{o}_1,\ldots,\mathrm{o}_m\}^{\vee},\mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = \{[\![\mathrm{o}_1]\!]^{\mathcal{J}},\ldots,[\![\mathrm{o}_m]\!]^{\mathcal{J}}\}$

Description: An extensional concept term is defined by its instances $\mathrm{o}_1,\ldots,\mathrm{o}_n$. Note that the instances mentioned in a **oneof** description, are full-fledged ABox objects and can have roles on their own, etc.

Example: european_country := **oneof**([france,germany,italy,uk]).

Idiosyncrasy: If intensional and extensional specifications are mixed (e.g., skandinavian_country := country **and oneof**([denmark, finland, norway, sweden]) it does not follow semantically that the constants mentioned extensionally are instances of the defined concept. Thus the above definition does not entail that denmark is a skandinavian_country, since it is not asserted that denmark is a country.

See also: **:/2**, **fillers**

or — Concept/Role Term

Synopsis:	Disjunction of concepts and roles
Syntax:	$\langle$*concept*$\rangle$ ::= $\langle$*concept*$\rangle$ **or** $\langle$*concept*$\rangle$ $\langle$*role*$\rangle$ ::= $\langle$*role*$\rangle$ **or** $\langle$*role*$\rangle$
Semantics:	$[\![\mathrm{t}_1 \sqcup \mathrm{t}_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = [\![\mathrm{t}_1, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} \cup [\![\mathrm{t}_2, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}}$
Description:	The instances of c_1 **or** c_2 are all objects which are either instances of c_1 or of c_2, or of both.
BACK:	The BACK system supports disjunction only in queries.
Example:	scientific_publication1 :<**ctop**. fiction1 :<**ctop**. publication1 :<scientific_publication1 **or** fiction1.
See also:	**and**, **not**

range — Role Term/Method

Synopsis: Restricts and retrieves the range of a role.

Syntax: $\langle$*role*$\rangle$ ::= **range**($\langle$*concept*$\rangle$)

Semantics: $[\![\mathrm{r}|_{\mathrm{c}}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} = [\![\mathrm{r}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}} \cap (D \times [\![\mathrm{c}, \mathrm{s}]\!]^{\mathcal{J},\mathcal{W}})$

Description: **range** is a role operator that restricts the fillers of a role to instances of a given concept. A role can be inverted only if its range is a “real” concept, i.e. not a **number** or a **string**. Defined role introductions, containing only a range restriction, i.e. r := **range**(c) are forbidden.

The method **range** retrieves the range of a role.

Example: publication_year :< **domain**(publication) **and range(number) and feature**.

See also: **comp**, **inv**, **domain**

satisfies — Ask Expression

Synopsis: Tests whether a term satisfies a filter.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
⟨*ask-expression*⟩	::=	**satisfies**(⟨*term*⟩,⟨*filter-list*⟩)

Description: **satisfies** performs a boolean test to determine whether the term given as argument satisfies the filter.

The **box** option is not meaningul for this ask expression.

Example: **flexask**(satisfies(european_conferenc,geo_type_conf)).

See also: **flexask**, **flexget**, **filter**

some — Concept Term

Synopsis: (Qualifying) existence restriction.

Syntax: $\langle$*concept*$\rangle$ ::= **some**($\langle$*role*$\rangle$,$\langle$*conceptual-type*$\rangle$)
| **some**($\langle$*role*$\rangle$)

Description: There is at least one role-filler at role r, resp. there is at least one role-filler of type c at role r.

Example: book_author := **some**(has_written,book).

Idiosyncrasy: Internally, the restrictions ‘**some**(r)’ and ‘**some**(r,c)’ are transformed into ‘**atleast**(1,r)’ and ‘**atleast**(1,r,c)’, respectively.

See also: **all**, **no**, **atleast**, **the**, **exactly**, **atmost**

string Term

Synopsis: Built-in topmost string.

Syntax: ⟨*concept*⟩ ::= **string**

Description: **string** is the topmost string having all other strings as instances.

Idiosyncrasy: Strings in FLEX are arbitrary Prolog atoms and need to be enclosed – according to the Prolog convention – in single-quotes if they contain special characters.

subs Method

Synopsis: Retrieves the subsumees of a term.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**subs**

Description: The method **subs** is applicable to concepts and roles and retrieves their subsumees. Meaningful options are **box** and **filter**.

The option **box** can be used to take into account rules and weighted defaults. If no box is specified the subsumption hierarchy stemming from definitions is taken; if **box=ibox**, the subsumption hierarchy also takes into account rules; if **box=dbox** even defaults are taken into account.

If the option **filter** is used, only terms satisfying the filter are taken into account in the subsumption hierarchy.

Example: **flexget**(article,**subs**,Result).

```
% binds Result to
[cbot,conference_article,journal_article]
```

See also: **dir_subs**, **supers**, **subsumes**

subsumes — Ask Expression

Synopsis: Subsumption test.

Syntax:

⟨*interaction*⟩	::=	**flexask**(⟨*ask-expression*⟩[,box=⟨*box*⟩])
⟨*ask-expression*⟩	::=	**subsumes**(⟨*term*⟩ ⟨*term*⟩)

Description: This operator performs a boolean test whether the ⟨*term*⟩ contained in the first argument subsumes the ⟨*term*⟩ contained in the second argument. Actually it is equivalent to **?**<.

If no box is specified, the test is performed on the basis of the term definitions; if **box=ibox** rules are also taken into account; if **box=dbox** rules and weighted defaults are taken into acount.

Example: **flexask**(**subsumes**(book,novel))

See also: **?**<, **equivalent**, **flexask**

supers Method

Synopsis: Retrieves the subsumers of a term.

Syntax:

⟨*interaction*⟩	::=	**flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩	::=	**supers**

Description: The method **subs** is applicable to concepts and roles and retrieves their subsumers. Meaningful options are **box** and **filter**.

The option **box** can be used to take into account rules and weighted defaults. If no box is specified the subsumption hierarchy stemming from definitions is taken; if **box=ibox**, the subsumption hierarchy also takes into account rules; if **box=dbox** even defaults are taken into account.

If the option **filter** is used, only terms satisfying the filter are taken into account in the subsumption hierarchy.

Example: **flexget**(novel,**supers**,Result).

```
% binds Result to
[ctop,publication,fiction,book]
```

See also: **dir_supers**, **subs**, **subsumes**

term_valued Role Term

Syntax: ⟨*role*⟩ ::= **term_valued**

Synopsis: A feature taking terms as values.

Description: Term-valued features take terms, i.e. concepts or roles as fillers instead of objects. The following inferences are performed for term-valued features:

$$\begin{array}{rcl} & \rightharpoonup & \mathrm{f}_t\mathrm{:t}_1 \sqcap \mathrm{f}_t\mathrm{:t}_2 \doteq \mathrm{f}_t : (\mathrm{t}_1 \sqcap \mathrm{t}_2) \\ & \rightharpoonup & \mathrm{f}_t : \bot \doteq \bot \\ \mathrm{t}_1 \sqsubseteq \mathrm{t}_2 & \rightharpoonup & \mathrm{f}_t\mathrm{:t}_1 \sqsubseteq \mathrm{f}_t\mathrm{:t}_2 \end{array}$$

Example: has_topic :<**domain**(scientific_publication)
and term_valued.

See also: **:/,tvf_filler**

the — Concept Term

Synopsis: Uniqueness restriction.

Syntax: ⟨*concept*⟩ ::= **the**(⟨*role*⟩,⟨*conceptual-type*⟩)

Description: There is exactly one role-filler at role r, and this role-filler is of type c. This operator makes value restrictions for functional roles (features) more readable, since **all**(age,**ge**(18)) easily has the connotation that an object has more than one age.
Internally, such a restriction is transformed into a conjunction of an all restriction and minimum and maximum restrictions with value 1.

Example: sixties_publication := publication **and**
the(publication_year,1960..1969).

Idiosyncrasy: Internally, the restriction '**the**(r,c)' is transformed into '**atleast**(1,r) **and atmost**(1,r) **and all**(r,c)'.

See also: **all**, **some**, **no**

tvf_filler Method

Synopsis: Retrieves the term filling a term-valued feature.

Syntax: ⟨*interaction*⟩ ::= **flexget**(⟨*entity*⟩,⟨*method*⟩,[⟨*options*⟩,]VAR)
⟨*method*⟩ ::= **tvf_filler**(**feature**[,⟨*sit*-NAME⟩])

Description: To retrieve the filler of a term-valued featuer at an object or a concept, the method **tvf_filler** can be used. Depending on the specified option for **box**, rules (**box=ibox**) and defaults (**box=dbox**) are taken into account to determine the fillers.

Example: **flexget**(flex,**tvf_filler**(has_topic,**initial**),Result).

```
% binds Result to description_logics
```

See also: **term_valued**, **fillers**, **:/2**