Handbook of Knowledge Representation
Edited by F. van Harmelen, V. Lifschitz and B. Porter

DOI: 10.1016/S1574-6526(07)03003-9

Chapter 3

Description Logics

Franz Baader, Ian Horrocks, Ulrike Sattler

In this chapter we will introduce description logics, a family of logic-based knowledge representation languages that can be used to represent the terminological knowledge of an application domain in a structured way. We will first review their provenance and history, and show how the field has developed. We will then introduce the basic description logic $\mathcal{ALC}$ in some detail, including definitions of syntax, semantics and basic reasoning services, and describe important extensions such as inverse roles, number restrictions, and concrete domains. Next, we will discuss the relationship between description logics and other formalisms, in particular first order and modal logics; the most commonly used reasoning techniques, in particular tableau, resolution and automata based techniques; and the computational complexity of basic reasoning problems. After reviewing some of the most prominent applications of description logics, in particular ontology language applications, we will conclude with an overview of other aspects of description logic research, and with pointers to the relevant literature.

3.1 Introduction

Description logics (DLs) [14, 25, 50] are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. The name *description logics* is motivated by the fact that, on the one hand, the important notions of the domain are described by concept *descriptions*, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL; on the other hand, DLs differ from their predecessors, such as semantic networks and frames, in that they are equipped with a formal, *logic*-based semantics.

We will first illustrate some typical constructors by an example; formal definitions will be given in Section 3.2. Assume that we want to define the concept of "A man that is married to a doctor, and all of whose children are either doctors or professors." This concept can be described with the following concept description:

$$\mathsf{Human} \sqcap \neg\mathsf{Female} \sqcap (\exists \mathsf{married}.\mathsf{Doctor}) \sqcap (\forall \mathsf{hasChild}.(\mathsf{Doctor} \sqcup \mathsf{Professor})).$$

This description employs the Boolean constructors *conjunction* ($\sqcap$), which is interpreted as set intersection, *disjunction* ($\sqcup$), which is interpreted as set union, and *negation* ($\neg$), which is interpreted as set complement, as well as the *existential restriction* constructor ($\exists r.C$), and the *value restriction* constructor ($\forall r.C$). An individual, say Bob, belongs to $\exists$married.Doctor if there exists an individual that is married to Bob (i.e., is related to Bob via the married role) and is a doctor (i.e., belongs to the concept Doctor). Similarly, Bob belongs to $\forall$hasChild.(Doctor $\sqcup$ Professor) if all his children (i.e., all individuals related to Bob via the hasChild role) are either doctors or professors.

Concept descriptions can be used to build statements in a DL knowledge base, which typically comes in two parts: a terminological and an assertional one. In the *terminological* part, called the TBox, we can describe the relevant notions of an application domain by stating properties of concepts and roles, and relationships between them—it corresponds to the *schema* in a database setting. In its simplest form, a TBox statement can introduce a name (abbreviation) for a complex description. For example, we could introduce the name HappyMan as an abbreviation for the concept description from above:

$$\mathsf{HappyMan} \equiv \mathsf{Human} \sqcap \neg\mathsf{Female} \sqcap (\exists \mathsf{married.Doctor}) \sqcap (\forall \mathsf{hasChild.}(\mathsf{Doctor} \sqcup \mathsf{Professor})).$$

More expressive TBoxes allow the statement of more general *axioms* such as

$$\exists \mathsf{hasChild.Human} \sqsubseteq \mathsf{Human},$$

which says that only humans can have human children. Note that, in contrast to the abbreviation statement from above, this statement does not define a concept. It just constrains the way in which concepts and roles (in this case, Human and hasChild) can be interpreted.

Obviously, all the knowledge we have described in our example could easily be represented by formulae of first-order predicate logic (see also Section 3.3). The variable-free syntax of description logics makes TBox statements easier to read than the corresponding first-order formulae. However, the main reason for using DLs rather than predicate logic is that DLs are carefully tailored such that they combine interesting means of expressiveness with decidability of the important reasoning problems (see below).

The *assertional* part of the knowledge base, called the ABox, is used to describe a concrete situation by stating properties of individuals—it corresponds to the *data* in a database setting. For example, the assertions

$$\mathsf{HappyMan(BOB)}, \quad \mathsf{hasChild(BOB, MARY)}, \quad \neg\mathsf{Doctor(MARY)}$$

state that Bob belongs to the concept HappyMan, that Mary is one of his children, and that Mary is not a doctor. Modern DL systems all employ this kind of restricted ABox formalism, which basically can be used to state ground facts. This differs from the use of the ABox in the early DL system KRYPTON [38], where ABox statements could be arbitrary first-order formulae. The underlying idea was that the ABox could then be used to represent knowledge that was not expressible in the restricted TBox formalism of KRYPTON, but this came with a cost: reasoning about ABox knowledge

required the use of a general theorem prover, which was quite inefficient and could lead to non-termination of the reasoning procedure.

Modern description logic systems provide their users with reasoning services that can automatically deduce implicit knowledge from the explicitly represented knowledge, and always yield a correct answer in finite time. In contrast to the database setting, such inference capabilities take into consideration *both* the terminological statements (schema) *and* the assertional statements (data). The *subsumption* algorithm determines subconcept-superconcept relationships: C is subsumed by D if all instances of C are necessarily instances of D, i.e., the first description is always interpreted as a subset of the second description. For example, given the definition of HappyMan from above plus the axiom $\mathsf{Doctor} \sqsubseteq \mathsf{Human}$, which says that all doctors are human, HappyMan is subsumed by $\exists \mathsf{married}.\mathsf{Human}$—since instances of HappyMan are married to some instance of Doctor, and all instances of Doctor are also instances of Human. The *instance* algorithm determines instance relationships: the individual i is an instance of the concept description C if i is always interpreted as an element of the interpretation of C. For example, given the assertions from above and the definition of HappyMan, MARY is an instance of Professor (because BOB is an instance of HappyMan, so all his children are either Doctors or Professors, MARY is a child of BOB, and MARY is not a Doctor). The *consistency* algorithm determines whether a knowledge base (consisting of a set of assertions and a set of terminological axioms) is non-contradictory. For example, if we add $\neg\mathsf{Professor}(\mathsf{MARY})$ to the three assertions from above, then the knowledge base containing these assertions together with the definition of HappyMan from above is inconsistent.

In a typical application, one would start building the TBox, making use of the reasoning services provided to ensure that all concepts in it are satisfiable, i.e., are not subsumed by the bottom concept, which is always interpreted as the empty set. Moreover, one would use the subsumption algorithm to compute the subsumption hierarchy, i.e., to check, for each pair of concept names, whether one is subsumed by the other. This hierarchy would then be inspected to make sure that it coincides with the intention of the modeler. Given, in addition, an ABox, one would first check for its consistency with the TBox and then, for example, compute the most specific concept(s) that each individual is an instance of (this is often called *realizing* the ABox). We could also use a concept description as a query, i.e., we could ask the DL system to identify all those individuals that are instances of the given, possibly complex, concept description.

In order to ensure a reasonable and predictable behavior of a DL system, these inference problems should at least be decidable for the DL employed by the system, and preferably of low complexity. Consequently, the expressive power of the DL in question must be restricted in an appropriate way. If the imposed restrictions are too severe, however, then the important notions of the application domain can no longer be expressed. Investigating this trade-off between the expressivity of DLs and the complexity of their inference problems has been one of the most important issues in DL research. This investigation has included both theoretical research, e.g., determining the worst case complexities for various DLs and reasoning problems, and practical research, e.g., developing systems and optimization techniques, and empirically evaluating their behavior when applied to benchmarks and used in various applications. The emphasis on decidable formalisms of restricted expressive power is also the reason why a great variety of extensions of basic DLs have been considered. Some of

these extensions leave the realm of classical first-order predicate logic, such as DLs with modal and temporal operators, fuzzy DLs, and probabilistic DLs (see [22] for details), but the goal of this research was still to design decidable extensions. If an application requires more expressive power than can be supplied by a decidable DL, then one usually embeds the DL into an application program or another KR formalism (see Section 3.8) rather than using an undecidable DL.

In the remainder of this section we will first give a brief overview of the history of DLs, and then describe the structure of this chapter. Research in Description Logics can be roughly classified into the following phases.

Phase 0 (1965–1980) is the pre-DL phase, in which *semantic networks* [138] and *frames* [122] were introduced as specialized approaches for representing knowledge in a structured way, and then criticized because of their lack of a formal semantics [163, 35, 84, 85]. An approach to overcome these problems was Brachman's *structured inheritance networks* [36], which were realized in the system KL-ONE, the first DL system.

Phase 1 (1980–1990) was mainly concerned with implementation of systems, such as KL-ONE, K-REP, KRYPTON, BACK, and LOOM [41, 119, 38, 137, 118]. These systems employed so-called *structural subsumption algorithms*, which first normalize the concept descriptions, and then recursively compare the syntactic structure of the normalized descriptions [126]. These algorithms are usually relatively efficient (polynomial), but they have the disadvantage that they are complete only for very inexpressive DLs, i.e., for more expressive DLs they cannot detect all subsumption/instance relationships. During this phase, the first logic-based accounts of the semantics of the underlying representation formalisms were given [38, 39], which made formal investigations into the complexity of reasoning in DLs possible. For example, in [39] it was shown that seemingly small additions to the expressive power of the representation formalism can cause intractability of the subsumption problem. In [148] it was shown that subsumption in the representation language underlying KL-ONE is even undecidable, and in [127] it was shown that the use of a TBox formalism that allows the introduction of abbreviations for complex descriptions makes subsumption intractable if the underlying DL has the constructors conjunction and value restriction (these constructors were supported by all the DL systems available at that time). As a reaction to these negative complexity results, the implementors of the CLASSIC system (the first industrial-strength DL system) carefully restricted the expressive power of their DL [135, 37].

Phase 2 (1990–1995) started with the introduction of a new algorithmic paradigm into DLs, so-called *tableau based algorithms* [149, 63, 89]. They work on propositionally closed DLs (i.e., DLs with all Boolean operators), and are complete also for expressive DLs. To decide the consistency of a knowledge base, a tableau based algorithm tries to construct a model of it by structurally decomposing the concepts in the knowledge base, thus inferring new constraints on the elements of this model. The algorithm either stops because all attempts to build a model failed with obvious contradictions, or it stops with a "canonical" model. Since, in propositionally closed DLs, the subsumption

and the instance problem can be reduced to consistency, a consistency algorithm can solve all the inference problems mentioned above. The first systems employing such algorithms (KRIS and CRACK) demonstrated that optimized implementations of these algorithms led to an acceptable behavior of the system, even though the worst-case complexity of the corresponding reasoning problems is no longer in polynomial time [18, 44]. This phase also saw a thorough analysis of the complexity of reasoning in various DLs [63, 64, 62], and the important observation that DLs are very closely related to modal logics [144].

Phase 3 (1995–2000) is characterized by the development of inference procedures for very expressive DLs, either based on the tableau approach [100, 92], or on a translation into modal logics [57, 58, 56, 59]. Highly optimized systems (FaCT, RACE, and DLP [95, 80, 133]) showed that tableau-based algorithms for expressive DLs led to a good practical behavior of the system even on (some) large knowledge bases. In this phase, the relationship to modal logics [57, 146] and to decidable fragments of first-order logic [33, 129, 79, 77, 78] was also studied in more detail, and applications in databases (like schema reasoning, query optimization, and integration of databases) were investigated [45, 47, 51].

We are now in *Phase 4*, where the results from the previous phases are being used to develop industrial strength DL systems employing very expressive DLs, with applications like the Semantic Web or knowledge representation and integration in medical- and bio-informatics in mind. On the academic side, the interest in less expressive DLs has been revived, with the goal of developing tools that can deal with very large terminological and/or assertional knowledge bases [6, 23, 53, 1].

The structure of the remainder of the chapter is as follows. In Section 3.2 we introduce the syntax and semantics of the prototypical DL $\mathcal{ALC}$, and some important extensions of $\mathcal{ALC}$. In Section 3.3 we discuss the relationship between DLs and other logical formalisms. In Section 3.4 we describe tableau-based reasoning techniques for $\mathcal{ALC}$, and in Section 3.5 we investigate the computation complexity of reasoning in $\mathcal{ALC}$. In Section 3.6 we introduce other reasoning techniques that can be used for DLs. In Section 3.7 we discuss the use of DLs in ontology language applications. Finally, in Section 3.8, we sketch important areas of DL research that have not been mentioned so far, and provide pointers to the literature.

Although we have endeavored to cover the most important areas of DL research, we have decided to treat some areas in more detail rather than giving a comprehensive survey of the whole field. Readers seeking such a survey are directed to [14].

3.2 A Basic DL and its Extensions

In this section we will define the syntax and semantics of the basic DL $\mathcal{ALC}$, and the most widely used DL reasoning services. We will also introduce important extensions to $\mathcal{ALC}$, including inverse roles, number restrictions, and concrete domains. The name $\mathcal{ALC}$ stands for "*A*ttributive concept *L*anguage with *C*omplements". It was first introduced in [149], where also a first naming scheme for DLs was proposed: starting from

a basic DL $\mathcal{AL}$, the addition of a constructors is indicated by appending a corresponding letter; e.g., $\mathcal{ALC}$ is obtained from $\mathcal{AL}$ by adding the complement operator ($\neg$) and $\mathcal{ALE}$ is obtained from $\mathcal{AL}$ by adding existential restrictions ($\exists r.C$) (for more details on such naming schemes for DLs, see [10]).

3.2.1 Syntax and Semantics of $\mathcal{ALC}$

In the following, we give formal definitions of the syntax and semantics of the constructors that we have described informally in the introduction. The DL that includes just this set of constructors (i.e., conjunction, disjunction, negation, existential restriction and value restriction) is called $\mathcal{ALC}$.

Definition 3.1 ($\mathcal{ALC}$ syntax). *Let N_C be a set of* concept names *and N_R be a set of* role names. *The set of $\mathcal{ALC}$-*concept descriptions *is the smallest set such that*

1. $\top$, $\bot$, *and every concept name $A \in N_C$ is an $\mathcal{ALC}$-concept description,*
2. *if C and D are $\mathcal{ALC}$-concept descriptions and $r \in N_R$, then $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall r.C$, and $\exists r.C$ are $\mathcal{ALC}$-concept descriptions.*

In the following, we will often use "$\mathcal{ALC}$-concept" instead of "$\mathcal{ALC}$-concept description". The semantics of $\mathcal{ALC}$ (and of DLs in general) is given in terms of *interpretations*.

Definition 3.2 ($\mathcal{ALC}$ semantics). *An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$, called the* domain *of $\mathcal{I}$, and a function $\cdot^{\mathcal{I}}$ that maps every $\mathcal{ALC}$-concept to a subset of $\Delta^{\mathcal{I}}$, and every role name to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ such that, for all $\mathcal{ALC}$-concepts C, D and all role names r,*

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}, \qquad \bot^{\mathcal{I}} = \emptyset,$$
$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}, \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}, \qquad \neg C^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$$
$$(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{There is some } y \in \Delta^{\mathcal{I}} \text{ with } \langle x, y\rangle \in r^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\},$$
$$(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid \text{For all } y \in \Delta^{\mathcal{I}}, \text{ if } \langle x, y\rangle \in r^{\mathcal{I}}, \text{ then } y \in C^{\mathcal{I}}\}.$$

We say that $C^{\mathcal{I}}$ ($r^{\mathcal{I}}$) is the extension of the concept C (role name r) in the interpretation $\mathcal{I}$. If $x \in C^{\mathcal{I}}$, then we say that x is an instance of C in $\mathcal{I}$.

As mentioned in the introduction, a DL knowledge base (KB) is made up of two parts, a terminological part (called the TBox) and an assertional part (called the ABox), each part consisting of a set of axioms. The most general form of TBox axioms are so-called general concept inclusions.

Definition 3.3. *A* general concept inclusion (*GCI*) *is of the form $C \sqsubseteq D$, where C, D are $\mathcal{ALC}$-concepts. A finite set of GCIs is called a TBox. An interpretation $\mathcal{I}$ is a* model *of a GCI $C \sqsubseteq D$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; $\mathcal{I}$ is a model of a TBox $\mathcal{T}$ if it is a model of every GCI in $\mathcal{T}$.*

We use $C \equiv D$ as an abbreviation for the symmetrical pair of GCIs $C \sqsubseteq D$ and $D \sqsubseteq C$.

An axiom of the form $A \equiv C$, where A is a concept name, is called a *definition*. A TBox $\mathcal{T}$ is called *definitorial* if it contains only definitions, with the additional restriction that (i) $\mathcal{T}$ contains at most one definition for any given concept name, and (ii) $\mathcal{T}$ is acyclic, i.e., the definition of any concept A in $\mathcal{T}$ does not refer (directly or indirectly) to A itself. Definitorial TBoxes are also called *acyclic* TBoxes in the literature. Given a definitorial TBox $\mathcal{T}$, concept names occurring on the left-hand side of such a definition are called *defined* concepts, whereas the others are called *primitive* concepts. The name "definitorial" is motivated by the fact that, in such a TBox, the extensions of the defined concepts are uniquely determined by the extensions of the primitive concepts and the role names. From a computational point of view, definitorial TBoxes are interesting since they may allow for the use of simplified reasoning techniques (see Section 3.4), and reasoning with respect to such TBoxes is often of a lower complexity than reasoning with respect to a general TBox (see Section 3.5).

The ABox can contain two kinds of axiom, one for asserting that an individual is an instance of a given concept, and the other for asserting that a pair of individuals is an instance of a given role name.

Definition 3.4. *An* assertional axiom *is of the form $x : C$ or $(x, y) : r$, where C is an $\mathcal{ALC}$-concept, r is a role name, and x and y are individual names. A finite set of assertional axioms is called an* ABox. *An interpretation $\mathcal{I}$ is a model of an assertional axiom $x : C$ if $x^{\mathcal{I}} \in C^{\mathcal{I}}$, and $\mathcal{I}$ is a model of an assertional axiom $(x, y) : r$ if $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$; $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ if it is a model of every axiom in $\mathcal{A}$.*

Several other notations for writing ABox axioms can be found in the literature, e.g., $C(x)$, $r(x, y)$ and $\langle x, y \rangle : r$.

Definition 3.5. *A* knowledge base (*KB*) *is a pair $(\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox. An interpretation $\mathcal{I}$ is a model of a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\mathcal{I}$ is a model of $\mathcal{T}$ and $\mathcal{I}$ is a model of $\mathcal{A}$.*

We will write $\mathcal{I} \models \mathcal{K}$ (resp. $\mathcal{I} \models \mathcal{T}$, $\mathcal{I} \models \mathcal{A}$, $\mathcal{I} \models a$) to denote that $\mathcal{I}$ is a model of a KB $\mathcal{K}$ (resp., TBox $\mathcal{T}$, ABox $\mathcal{A}$, axiom a).

3.2.2 Important Inference Problems

We define inference problems with respect to a KB consisting of a TBox and an ABox. Later on, we will also consider special cases where the TBox or/and ABox is empty, or where the TBox satisfies additional restrictions, such as being definitorial.

Definition 3.6. *Given a KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where $\mathcal{T}$ is a TBox and $\mathcal{A}$ is an ABox, $\mathcal{K}$ is called* consistent *if it has a model. A concept C is called* satisfiable with respect to $\mathcal{K}$ *if there is a model $\mathcal{I}$ of $\mathcal{K}$ with $C^{\mathcal{I}} \neq \emptyset$. Such an interpretation is called a* model of C *with respect to $\mathcal{K}$. The concept D* subsumes *the concept C with respect to $\mathcal{K}$ (written $\mathcal{K} \models C \sqsubseteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for all models $\mathcal{I}$ of $\mathcal{K}$. Two concepts C, D are* equivalent *with respect to $\mathcal{K}$ (written $\mathcal{K} \models C \equiv D$) if they subsume each other with*

respect to $\mathcal{K}$. *An individual* a *is an* instance of a concept C with respect to $\mathcal{K}$ (*written* $\mathcal{K} \models a : C$) *if* $a^{\mathcal{I}} \in C^{\mathcal{I}}$ *holds for all models* $\mathcal{I}$ *of* $\mathcal{K}$. *A pair of individuals* (a, b) *is an* instance of a role name r with respect to $\mathcal{K}$ (*written* $\mathcal{K} \models (a, b) : r$) *if* $\langle a^{\mathcal{I}}, b^{\mathcal{I}} \rangle \in r^{\mathcal{I}}$ *holds for all models* $\mathcal{I}$ *of* $\mathcal{K}$.

For a DL providing all the Boolean operators, like $\mathcal{ALC}$, all of the above reasoning problems can be reduced to KB consistency. For example, $(\mathcal{T}, \mathcal{A}) \models a : C$ iff $(\mathcal{T}, \mathcal{A} \cup \{a : \neg C\})$ is inconsistent. We will talk about satisfiability (resp., subsumption and equivalence) with respect to a TBox $\mathcal{T}$, meaning satisfiability (resp., subsumption and equivalence) with respect to the KB $(\mathcal{T}, \emptyset)$. This is often referred to as *terminological* reasoning. In many cases (e.g., in the case of $\mathcal{ALC}$), the ABox has no influence on terminological reasoning, i.e., satisfiability (resp., subsumption and equivalence) with respect to $(\mathcal{T}, \mathcal{A})$ coincides with satisfiability (resp., subsumption and equivalence) with respect to $\mathcal{T}$, as long as the ABox $\mathcal{A}$ is consistent (i.e., has a model).

3.2.3 Important Extensions to $\mathcal{ALC}$

One prominent application of DLs is as the formal foundation for ontology languages. Examples of DL based ontology languages include OIL [69], DAML + OIL [97, 98], and OWL [134], a recently emerged ontology language standard developed by the W3C Web-Ontology Working Group.[1]

High quality ontologies are crucial for many applications, and their construction, integration, and evolution greatly depends on the availability of a well-defined semantics and powerful reasoning tools. Since DLs provide for both, they should be ideal candidates for ontology languages. That much was already clear ten years ago, but at that time there was a fundamental mismatch between the expressive power and the efficiency of reasoning that DL systems provided, and the expressivity and the large knowledge bases that users needed [67]. Through basic research in DLs over the last 10–15 years, as summarized in the introduction, this gap between the needs of ontologist and the systems that DL researchers provide has finally become narrow enough to build stable bridges. In particular, $\mathcal{ALC}$ has been extended with several features that are important in an ontology language, including (qualified) number restrictions, inverse roles, transitive roles, subroles, concrete domains, and nominals.

With *number restrictions*, it is possible to describe the number of relationships of a particular type that individuals can participate in. For example, we may want to say that a person can be married to at most one other individual:

$$\mathsf{Person} \sqsubseteq \leqslant 1\, \mathsf{married},$$

and we may want to extend our definition of HappyMan to include the fact that instances of HappyMan have between two and four children:

$$\begin{aligned} \mathsf{HappyMan} \equiv\ & \mathsf{Human} \sqcap \neg \mathsf{Female} \sqcap (\exists \mathsf{married}.\mathsf{Doctor}) \\ & \sqcap (\forall \mathsf{hasChild}.(\mathsf{Doctor} \sqcup \mathsf{Professor})) \\ & \sqcap \geqslant 2\, \mathsf{hasChild} \sqcap \leqslant 4\, \mathsf{hasChild}. \end{aligned}$$

[1] http://www.w3.org/2001/sw/WebOnt/.

With *qualified number restrictions*, we can additionally describe the type of individuals that are counted by a given number restriction. For example, using qualified number restrictions, we could further extend our definition of HappyMan to include the fact that instances of HappyMan have at least two children who are doctors:

$$\begin{aligned}\mathsf{HappyMan} \equiv\ & \mathsf{Human} \sqcap \neg\mathsf{Female} \sqcap (\exists \mathsf{married}.\mathsf{Doctor}) \\ & \sqcap (\forall \mathsf{hasChild}.(\mathsf{Doctor} \sqcup \mathsf{Professor})) \\ & \sqcap \geqslant 2\, \mathsf{hasChild}.\mathsf{Doctor} \sqcap \leqslant 4\, \mathsf{hasChild}.\end{aligned}$$

With *inverse roles*, *transitive roles*, and *subroles* [100] we can, in addition to hasChild, also use its inverse hasParent, specify that hasAncestor is transitive, and specify that hasParent is a subrole of hasAncestor.

Concrete domains [16, 115] integrate DLs with concrete sets such as the real numbers, integers, or strings, as well as concrete predicates defined on these sets, such as numerical comparisons (e.g., $\leqslant$), string comparisons (e.g., isPrefixOf), or comparisons with constants (e.g., $\leqslant 17$). This supports the modeling of concrete properties of abstract objects such as the age, the weight, or the name of a person, and the comparison of these concrete properties. Unfortunately, in their unrestricted form, concrete domains can have dramatic effects on the decidability and computational complexity of the underlying DL [17, 115]. For this reason, a more restricted form of concrete domain, known as *datatypes* [101], is often used in practice.

The *nominal* constructor allows us to use individual names also within concept descriptions: if a is an individual name, then $\{a\}$ is a concept, called a nominal, which is interpreted by a singleton set. Using the individual Turing, we can describe all those computer scientists that have met Turing by $\mathsf{CScientist} \sqcap \exists \mathsf{hasMet}.\{\mathsf{Turing}\}$. The so-called "one-of" constructor extends the nominal constructor to a finite set of individuals. In the presence of disjunction, it can, however, be expressed using nominals: $\{a_1, \ldots, a_n\}$ is equivalent to $\{a_1\} \sqcup \cdots \sqcup \{a_n\}$. The presence of nominals can have dramatic effects on the complexity of reasoning [159].

An additional comment on the naming of DLs is in order. Recall that the name given to a particular DL usually reflects its expressive power, with letters expressing the constructors provided. For expressive DLs, starting with the basic DL $\mathcal{AL}$ would lead to quite long names. For this reason, the letter $\mathcal{S}$ is often used as an abbreviation for the "basic" DL consisting of $\mathcal{ALC}$ extended with transitive roles (which in the $\mathcal{AL}$ naming scheme would be called $\mathcal{ALC}_{R^+}$).[2] The letter $\mathcal{H}$ represents subroles (role $\mathcal{H}$ierarchies), $\mathcal{O}$ represents nominals (n$\mathcal{O}$minals), $\mathcal{I}$ represents inverse roles ($\mathcal{I}$nverse), $\mathcal{N}$ represent number restrictions ($\mathcal{N}$umber), and $\mathcal{Q}$ represent qualified number restrictions ($\mathcal{Q}$ualified). The integration of a concrete domain/datatype is indicated by appending its name in parenthesis, but sometimes a "generic" **D** is used to express that some concrete domain/datatype has been integrated. The DL corresponding to the OWL DL ontology language includes all of these constructors and is therefore called $\mathcal{SHOIN}(\mathbf{D})$.

[2] The use of $\mathcal{S}$ is motivated by the close connection between this DL and the modal logic **S4**.

3.3 Relationships with other Formalisms

In this section, we discuss the relationships between DLs and predicate logic, and between DLs and Modal Logic. This is intended for readers who are familiar with these logics; those not familiar with these logics might want to skip the following subsection(s), since we do not introduce modal or predicate logic here—we simply use standard terminology. Here, we only describe the relationship of the basic DL $\mathcal{ALC}$ and some of its extensions to these other logics (for a more detailed analysis, see [33] and Chapter 4 of [14]).

3.3.1 DLs and Predicate Logic

Most DLs can be seen as fragments of first-order predicate logic, although some provide operators such as transitive closure of roles or fixpoints that require second-order logic [33]. The main reason for using Description Logics rather than general first-order predicate logic when representing knowledge is that most DLs are actually *decidable* fragments of first-order predicate logic, i.e., there are effective procedures for deciding the inference problems introduced above.

Viewing role names as binary relations and concept names as unary relations, we define two translation functions, π_x and π_y, that inductively map $\mathcal{ALC}$-concepts into first order formulae with one free variable, x or y:

$$\begin{aligned}
\pi_x(A) &= A(x), & \pi_y(A) &= A(y),\\
\pi_x(C \sqcap D) &= \pi_x(C) \wedge \pi_x(D), & \pi_y(C \sqcap D) &= \pi_y(C) \wedge \pi_y(D),\\
\pi_x(C \sqcup D) &= \pi_x(C) \vee \pi_x(D), & \pi_y(C \sqcup D) &= \pi_y(C) \vee \pi_y(D),\\
\pi_x(\exists r.C) &= \exists y.r(x, y) \wedge \pi_y(C), & \pi_y(\exists r.C) &= \exists x.r(y, x) \wedge \pi_x(C),\\
\pi_x(\forall r.C) &= \forall y.r(x, y) \Rightarrow \pi_y(C), & \pi_y(\forall r.C) &= \forall x.r(y, x) \Rightarrow \pi_x(C).
\end{aligned}$$

Given this, we can translate a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ as follows, where $\psi[x/a]$ denotes the formula obtained from ψ by replacing all free occurrences of x with a:

$$\begin{aligned}
\pi(\mathcal{T}) &= \bigwedge_{C \sqsubseteq D \in \mathcal{T}} \forall x.(\pi_x(C) \Rightarrow \pi_x(D)),\\
\pi(\mathcal{A}) &= \bigwedge_{a:C \in \mathcal{A}} \pi_x(C)[x/a] \wedge \bigwedge_{(a,b):r \in \mathcal{A}} r(a, b).
\end{aligned}$$

This translation preserves the semantics: we can obviously view DL interpretations as first-order interpretations and vice versa, and it is easy to show that the translation preserves models. As an easy consequence, we have that reasoning in DLs corresponds to first-order inference:

Theorem 3.1. *Let $(\mathcal{T}, \mathcal{A})$ be an $\mathcal{ALC}$-knowledge base, C, D possibly complex $\mathcal{ALC}$-concepts, and a an individual name. Then*

1. $(\mathcal{T}, \mathcal{A})$ *is consistent iff* $\pi(\mathcal{T}) \wedge \pi(\mathcal{A})$ *is consistent,*
2. $(\mathcal{T}, \mathcal{A}) \models C \sqsubseteq D$ *iff* $(\pi(\mathcal{T}) \wedge \pi(\mathcal{A})) \Rightarrow (\pi(\{C \sqsubseteq D\}))$ *is valid,*
3. $(\mathcal{T}, \mathcal{A}) \models a : C$ *iff* $(\pi(\mathcal{T}) \wedge \pi(\mathcal{A})) \Rightarrow (\pi(\{a : C\}))$ *is valid.*

This translation not only provides an alternative way of defining the semantics of $\mathcal{ALC}$, but also tells us that all the introduced reasoning problems for $\mathcal{ALC}$ knowledge

bases are decidable. In fact, the translation of a knowledge base uses only variables x and y, and thus yields a formula in the *two variable fragment of first-order logic*, which is known to be decidable in non-deterministic exponential time [79]. Alternatively, we can use the fact that this translation uses quantification only in a restricted way, and therefore yields a formula in the *guarded fragment* [2], which is known to be decidable in deterministic exponential time [78]. Thus, the exploration of the relationship between DLs and first-order logics even gives us upper complexity bounds "for free". However, for $\mathcal{ALC}$ and also many other DLs, the upper bounds obtained this way are not necessarily optimal, which justifies the development of dedicated reasoning procedures for DLs.

The translation of more expressive DLs may be straightforward, or more difficult, depending on the additional constructs. Inverse roles can be captured easily in both the guarded and the two variable fragment by simply swapping the variable places; e.g., $\pi_x(\exists R^-.C) = \exists y.R(y,x) \wedge \pi_y(C)$. Number restrictions can be captured using (in)equality or so-called *counting quantifiers*. It is known that the two-variable fragment with counting quantifiers is still decidable in non-deterministic exponential time [130]. Transitive roles, however, cannot be expressed with two variables only, and the three variable fragment is known to be undecidable. The guarded fragment, when restricted carefully to the so-called *action guarded fragment* [75], can still capture a variety of features such as number restrictions, inverse roles, and fixpoints, while remaining decidable in deterministic exponential time.

3.3.2 DLs and Modal Logic

Description Logics are closely related to Modal Logics, yet they have been developed independently. This close relationship was discovered relatively late [144], but has since then been exploited quite successfully to transfer complexity and decidability results as well as reasoning techniques [145, 57, 90, 3]. It is not hard to see that $\mathcal{ALC}$-concepts can be viewed as syntactic variants of formulae of the (multi) modal logic **K**: Kripke structures can easily be viewed as DL interpretations and, conversely, DL interpretations as Kripke structures; we can then view concept names as propositional variables, and role names as modal parameters, and realize this correspondence through the rewriting $\leftrightsquigarrow$, which allows $\mathcal{ALC}$-concepts to be translated into modal formulae and conversely modal formulae into $\mathcal{ALC}$-concepts, as follows:

$\mathcal{ALC}$-concept		Modal **K** formula
A	$\leftrightsquigarrow$	a, for concept name A and propositional variable a,
$C \sqcap D$	$\leftrightsquigarrow$	$C \wedge D$,
$C \sqcup D$	$\leftrightsquigarrow$	$C \vee D$,
$\neg C$	$\leftrightsquigarrow$	$\neg C$,
$\forall r.C$	$\leftrightsquigarrow$	$[r]C$,
$\exists r.C$	$\leftrightsquigarrow$	$\langle r \rangle C$.

Let us use $\dot{C}$ for the modal formula obtained by rewriting the $\mathcal{ALC}$-concept C. The translation of DL knowledge bases is slightly more tricky: a TBox $\mathcal{T}$ is satisfied only in those structures where, for each $C \sqsubseteq D$, $\neg\dot{C} \vee \dot{D}$ holds *globally*, i.e., in each world of our Kripke structure (or, equivalently, in each element of our interpretation domain). We can express this using the universal modality, that is, a special modal parameter

U that is interpreted as the total relation in all Kripke structures. Before we discuss ABoxes, let us first state the properties of our correspondence so far.

Theorem 3.2. *Let $\mathcal{T}$ be an $\mathcal{ALC}$-TBox and E, F possibly complex $\mathcal{ALC}$-concepts. Then*

1. *F is satisfiable with respect to $\mathcal{T}$* *iff* $\dot{F} \wedge \bigwedge_{C \sqsubseteq D \in \mathcal{T}} [U](\neg \dot{C} \vee \dot{D})$ *is satisfiable,*
2. $\mathcal{T} \models E \sqsubseteq F$ *iff* $(\bigwedge_{C \sqsubseteq D \in \mathcal{T}} [U](\neg \dot{C} \vee \dot{D})) \wedge \dot{E} \wedge \neg \dot{F}$ *is unsatisfiable.*

Like TBoxes, ABoxes do not have a direct correspondence in modal logic, but they can be seen as a special case of a modal logic constructor, namely *nominals*. These are special propositional variables that hold in exactly one world; they are the basic ingredient of *hybrid logics* [4], and usually come with a special modality, the @-operator, that allows one to refer to the (only) world in which the nominal a holds. For example, $@_a \psi$ holds if, in the world where a holds, ψ holds as well. Hence an ABox assertion of the form $a : C$ corresponds to the modal formula $@_a \dot{C}$, and an ABox assertion $(a, b) : r$ corresponds to $@_a \langle r \rangle b$. In this latter formula, we see that nominals can act both as a parameter to the @ operator, like a, and as a propositional variables, like b. Please note that the usage of individual names in ABoxes corresponds to formulae where nominals are used in a rather restricted form only—some DLs, such as $\mathcal{SHOIN}$ or $\mathcal{SHOIQ}$, allow for a more general use of nominals, which is normally indicated by the letter $\mathcal{O}$ in a DL's name.

As in the case of first-order logic, some DL constructors have close relatives in modal logics and some do not. Number restrictions correspond to so-called *graded modalities* [70], which in modal logic received only limited attention until the connection with DLs was found. In some variants of propositional dynamic logic [71], a modal logic for reasoning about programs, we find *deterministic programs*, which correspond to (unqualified) number restrictions of the form $\leqslant 1R.\top$ [29]. Similarly, we find there *converse programs*, which correspond to *inverse roles*, and *regular expressions of programs*, which correspond to roles built using transitive-reflexive closure, union, and composition.

3.4 Tableau Based Reasoning Techniques

A variety of reasoning techniques can be used to solve the reasoning problems introduced in Section 3.2. These include resolution based approaches [102, 104], automata based approaches [49, 161], and structural approaches (for sub-Boolean DLs) [6]. The most widely used technique, however, is the tableau based approach first introduced by Schmidt-Schauß and Smolka [149]. In this section, we described this technique for the case of our basic DL $\mathcal{ALC}$.

3.4.1 A Tableau Algorithm for $\mathcal{ALC}$

We will concentrate on knowledge base consistency because, as we have seen in Section 3.2, this is a very general problem to which many others can be reduced. For

example, given a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, a concept C is subsumed by a concept D with respect to $\mathcal{K}$ ($\mathcal{K} \models C \sqsubseteq D$) iff $(\mathcal{T}, \mathcal{A} \cup \{x : (C \sqcap \neg D)\})$ is not consistent, where x is a new individual name (i.e., one that does not occur in $\mathcal{K}$). For $\mathcal{ALC}$ with a general TBox, i.e., one where the TBox is not restricted to contain only definitorial axioms (see Section 3.2), this problem is known to be EXPTIME-complete [144].

The tableau based decision procedure for the consistency of general $\mathcal{ALC}$ knowledge bases sketched below (and described in more detail in [12, 14]), runs in worst-case *non-deterministic* double exponential time.[3] However, according to the current state of the art, procedures such as this work well in practice, and are the basis for highly optimized implementations of DL systems such as FaCT [95], FaCT++ [160], RACER [81] and Pellet [151].

Given a knowledge base $(\mathcal{T}, \mathcal{A})$, we can assume, without loss of generality, that all of the concepts occurring in $\mathcal{T}$ and $\mathcal{A}$ are in *negation normal form* (NNF), i.e., that negation is applied only to concept names. An arbitrary $\mathcal{ALC}$ concept can be transformed to an equivalent one in NNF by pushing negations inwards using a combination of de Morgan's laws and the duality between existential and universal restrictions ($\neg\exists r.C \equiv \forall r.\neg C$ and $\neg\forall r.C \equiv \exists r.\neg C$). For example, the concept $\neg(\exists r.A \sqcap \forall s.B)$, where A, B are concept names, can be transformed to the equivalent NNF concept $(\forall r.\neg A) \sqcup (\exists s.\neg B)$. For a concept C, we will use $\dot{\neg}C$ to denote the NNF of $\neg C$.

The idea behind the algorithm is that it tries to prove the consistency of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ by constructing (a representation of) a model of $\mathcal{K}$. It does this by starting from the concrete situation described in $\mathcal{A}$, and explicating additional constraints on the model that are implied by the concepts in $\mathcal{A}$ and the axioms in $\mathcal{T}$. Since $\mathcal{ALC}$ has a so-called forest model property, we can assume that this model has the form of a set of (potentially infinite) trees, the root nodes of which can be arbitrarily interconnected. If we want to obtain a decision procedure, we can only construct finite trees representing the (potentially) infinite ones (assuming that a model exists at all); this can be done such that the finite representation can be *unraveled* into an infinite forest model $\mathcal{I}$ of $(\mathcal{T}, \mathcal{A})$.

In order to construct such a finite representation, the algorithm works on a data structure called a *completion forest*. This consists of a labelled directed graph, each node of which is the root of a *completion tree*. Each node x in the completion forest (which is either a root node or a node in a completion tree) is labelled with a set of concepts $\mathcal{L}(x)$, and each edge $\langle x, y\rangle$ (which is either one between root nodes or one inside a completion tree) is labelled with a set of role names $\mathcal{L}(\langle x, y\rangle)$. If $\langle x, y\rangle$ is an edge in the completion forest, then we say that x is a predecessor of y (and that y is a successor of x); in case $\langle x, y\rangle$ is labelled with a set containing the role name r, then we say that y is an r-successor of x.

When started with a knowledge base $(\mathcal{T}, \mathcal{A})$, the completion forest $\mathcal{F}_\mathcal{A}$ is initialized such that it contains a root node x_a, with $\mathcal{L}(x_a) = \{C \mid a\colon C \in \mathcal{A}\}$, for each individual name a occurring in $\mathcal{A}$, and an edge $\langle x_a, x_b\rangle$, with $\mathcal{L}(\langle x_a, x_b\rangle) = \{r \mid (a, b)\colon r \in \mathcal{A}\}$, for each pair (a, b) of individual names for which the set $\{r \mid (a, b)\colon r \in \mathcal{A}\}$ is nonempty.

[3] This is due to the algorithm searching a tree of worst-case exponential depth. By re-using previously computed search results, a similar algorithm can be made to run in exponential time [66], but this introduces a considerable overhead which turns out to be not always useful in practice.

$\sqcap$-rule:	if 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, x is not blocked, and 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$ then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$-rule:	if 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, x is not blocked, and 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
$\exists$-rule:	if 1. $\exists r.C \in \mathcal{L}(x)$, x is not blocked, and 2. x has no r-successor y with $C \in \mathcal{L}(y)$, then create a new node y with $\mathcal{L}(\langle x, y\rangle) = \{r\}$ and $\mathcal{L}(y) = \{C\}$
$\forall$-rule:	if 1. $\forall r.C \in \mathcal{L}(x)$, x is not blocked, and 2. there is an r-successor y of x with $C \notin \mathcal{L}(y)$ then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$
$\sqsubseteq$-rule:	if 1. $C_1 \sqsubseteq C_2 \in \mathcal{T}$, x is not blocked, and 2. $C_2 \sqcup \dot{\neg} C_1 \notin \mathcal{L}(x)$ then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_2 \sqcup \dot{\neg} C_1\}$

Figure 3.1: The tableau expansion rules for $\mathcal{ALC}$.

The algorithm then applies so-called *expansion rules*, which syntactically decompose the concepts in node labels, either inferring new constraints for a given node, or extending the tree according to these constraints (see Fig. 3.1). For example, if $C_1 \sqcap C_2 \in \mathcal{L}(x)$, and either $C_1 \notin \mathcal{L}(x)$ or $C_2 \notin \mathcal{L}(x)$, then the $\sqcap$-rule adds both C_1 and C_2 to $\mathcal{L}(x)$; if $\exists r.C \in \mathcal{L}(x)$, and x does not yet have an r-successor with C in its label, then the $\exists$-rule generates a new r-successor node y of x with $\mathcal{L}(y) = \{C\}$. Note that the $\sqcup$-rule is different from the other rules in that it is *non-deterministic*: if $C_1 \sqcup C_2 \in \mathcal{L}(x)$ and neither $C_1 \in \mathcal{L}(x)$ nor $C_2 \in \mathcal{L}(x)$, then it adds *either* C_1 *or* C_2 to $\mathcal{L}(x)$. In practice this is the main source of complexity in tableau algorithms, because it may be necessary to explore all possible choices of rule applications.

The algorithm stops if it encounters a *clash*: a completion forest in which $\{A, \neg A\} \subseteq \mathcal{L}(x)$ for some node x and some concept name A. In this case, the completion forest contains an obvious inconsistency, and thus does not represent a model. If the algorithm stops without having encountered a clash, then the obtained completion forest yields a finite representation of a forest model, and the algorithm answers "$(\mathcal{T}, \mathcal{A})$ is consistent"; if none of the possible non-deterministic choices of the $\sqcup$-rule leads to such a representation of a forest model, i.e., all of them lead to a clash, then the algorithm answers "$(\mathcal{T}, \mathcal{A})$ is inconsistent".

Please note that we have two different kinds of non-determinism in this algorithm. The non-deterministic choice between the two disjuncts in the $\sqcup$-rule is "don't know" non-deterministic, i.e., if the first choice leads to a clash, then the second one must be explored. In contrast, the choice of which rule to apply next to a given completion forest is "don't care" non-deterministic, i.e., one can choose an arbitrary applicable rule without the need to backtrack and explore alternative choices.

It remains to explain the meaning of "blocked" in the formulation of the expansion rules. Without the $\sqsubseteq$-rule (i.e., in case the TBox is empty), the tableau algorithm for $\mathcal{ALC}$ would always terminate, even without blocking. In order to guarantee termination of the expansion process even in the presence of GCIs, the algorithm uses

a technique called *blocking*.[4] Blocking prevents application of expansion rules when the construction becomes repetitive; i.e., when it is obvious that the sub-tree rooted in some node x will be "similar" to the sub-tree rooted in some predecessor y of x. To be more precise, we say that a node y is an *ancestor* of a node x if they both belong to the same completion tree and either y is a predecessor of x, or there exists a predecessor z of x such that y is an ancestor of z. A node x is *blocked* if there is an ancestor y of x such that $\mathcal{L}(x) \subseteq \mathcal{L}(y)$ (in this case we say that y blocks x), or if there is an ancestor z of x such that z is blocked; if a node x is blocked and none of its ancestors is blocked, then we say that x is *directly blocked*. When the algorithm stops with a clash free completion forest, a branch that contains a directly blocked node x represents an infinite branch in the corresponding model having a regular structure that corresponds to an infinite repetition (or "unraveling") of the section of the graph between x and the node that blocks it (see Section 3.6.1).

Theorem 3.3. *The above algorithm is a decision procedure for the consistency of $\mathcal{ALC}$ knowledge bases.*

A complete proof of this theorem is beyond the scope of this chapter, and we will only sketch the idea behind the proof: the interested reader can refer to [12, 14] for more details. Firstly, it is easy to see that the algorithm terminates: expansion rule applications always extend node labels or add new nodes, and we can fix an upper bound on the size of node labels (they can only contain concepts that are derivable from the syntactic decomposition of concepts occurring in the input KB), on the fan-out of trees in the completion forest (a node can have at most one successor for each existential restriction occurring in its label), and on the length of their branches (due to blocking). Secondly, soundness follows from the fact that we can transform a fully expanded and clash free completion forest into a model of the input KB by "throwing away" all blocked nodes and "bending" each edge from a non-blocked into a blocked node to the node it is blocked by.[5] Finally, completeness follows from the fact that, given a model of the input KB, we could use it to guide applications of the $\sqcup$-rule so as to produce a fully expanded and clash free completion forest.

The procedure described above can be simplified if the TBox is definitorial, i.e., if it contains only unique and acyclic definitions (see Section 3.2). In this case, reasoning with a knowledge base can be reduced to the problem of reasoning with an ABox only (equivalently, a knowledge base with an empty TBox) by *unfolding* the concepts used in ABox axioms [126]: given a KB $(\mathcal{T}, \mathcal{A})$, where the definition $A \equiv C$ occurs in $\mathcal{T}$, all occurrences of A in $\mathcal{A}$ can be replaced with C. Repeated application of this procedure can be used to eliminate from $\mathcal{A}$ all those concept names for which there is a definition in $\mathcal{T}$. As mentioned above, when the TBox is empty the $\sqsubseteq$-rule is no longer required and blocking can be dispensed with. This is because the other rules only introduce concepts that are smaller than the concept triggering the rule application; we will come back to this in Section 3.5.1.

[4] In description logics, blocking was first employed in [8] in the context of an algorithm that can handle the transitive closure of roles, and was improved on in [13, 46, 12, 92].

[5] For $\mathcal{ALC}$, we can always construct a finite cyclical model in this way; for more expressive DLs, we may need different blocking conditions, and we may need to unravel such cycles in order to construct an infinite model.

It is easy to see that the above *static* unfolding procedure can lead to an exponential increase in the size of the ABox [126]. In general, this cannot be avoided since there are DLs where reasoning with respect to definitorial TBoxes is harder than without TBoxes [127, 114]. For $\mathcal{ALC}$, however, we can avoid an increase in the complexity of the algorithm by unfolding definitions not a priori, but only as required by the progress of the algorithm. This so-called *lazy* unfolding [15, 95, 114] is achieved by substituting the $\sqsubseteq$-rule by the following two $\equiv_i$-rules:

$\equiv_1$-rule: if 1. $A \equiv C \in \mathcal{T}$, $A \in \mathcal{L}(x)$,
2. and $C \notin \mathcal{L}(x)$,
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$;

$\equiv_2$-rule: if 1. $A \equiv C \in \mathcal{T}$, $\neg A \in \mathcal{L}(x)$,
2. and $\dot{\neg} C \notin \mathcal{L}(x)$,
then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{\dot{\neg} C\}$.

As in the case of static unfolding, blocking is not required: the acyclicity condition on the TBox means that if a concept C is added to $\mathcal{L}(x)$ as a result of an application of one of the $\equiv_i$-rules to the concept A or $\neg A$ and axiom $A \equiv C$, then further unfolding of C cannot lead to the introduction of another occurrence of A in the sub-tree below x.

The tableau algorithm can also be extended to deal with a wide range of other DLs, including those supporting, e.g., (qualified) number restrictions, inverse roles, transitive roles, subroles, concrete domains and nominals. Extending the algorithm to deal with such features is mainly a matter of adding expansion rules to deal with the new constructors (e.g., number restrictions), adding new clash conditions (e.g., to deal with obviously unsatisfiable number restrictions), and using a more sophisticated blocking condition in order to guarantee both termination and soundness when using the extended rule set.

3.4.2 Implementation and Optimization Techniques

Although reasoning in $\mathcal{ALC}$ (with respect to an arbitrary KB) is of a relatively high complexity (EXPTIME-complete), the pathological cases that lead to such high *worst case* complexity are rather artificial, and rarely occur in practice [127, 86, 154, 95]. Even in realistic applications, however, problems can occur that are much too hard to be solved by naive implementations of theoretical algorithms such as the one sketched in Section 3.4.1. Modern DL systems, therefore, include a wide range of optimization techniques, the use of which has been shown to improve *typical case* performance by several orders of magnitude [96]. These systems exhibit good typical case performance, and work well in realistic applications [15, 44, 95, 81, 133].

A detailed description of optimization techniques is beyond the scope of this chapter, and the interested reader is referred to Chapter 8 of [14] for further information. It will, however, be interesting to sketch a couple of the key techniques: absorption and dependency directed backtracking.

Absorption

Whereas definitorial TBoxes can be dealt with efficiently by using lazy unfolding (see Section 3.4.1 above), more general axioms are not amenable to this optimization technique. In particular, GCIs $C \sqsubseteq D$, where C is non-atomic, must be dealt with by explicitly making every individual in the model an instance of $D \sqcup \dot{\neg} C$ (see Fig. 3.1). Large numbers of such GCIs result in a very high degree of non-determinism due to