4

Reasoning in DLs with Tableau Algorithms

A variety of reasoning techniques can be used to solve (some of) the reasoning problems introduced in Chapter 2. These include resolution and consequence-based approaches (see Chapter 6), automata-based approaches (see Section 3.5) and query rewriting approaches (see Chapter 7). For reasoning with expressive DLs,[1] however, the most widely used technique is the tableau-based approach.

We will concentrate on knowledge base consistency because, as we saw in Theorem 2.17, 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 $\mathcal{K}$ ($\mathcal{K} \models C \sqsubseteq D$) if and only if $(\mathcal{T}, \mathcal{A} \cup \{x : C \sqcap \neg D\})$ is not consistent, where x is a new individual name (i.e., one that does not occur in $\mathcal{K}$). Similarly, an individual name a is an instance of a concept C with respect to $\mathcal{K}$ ($\mathcal{K} \models a : C$) if and only if $(\mathcal{T}, \mathcal{A} \cup \{a : \neg C\})$ is not consistent. In practice, highly optimised tableau algorithms for deciding knowledge base consistency form the core of several implemented systems, including FaCT, FaCT++, Pellet, Racer and Konclude.

In the following we will:

- describe the general principles of the tableau approach;
- present an algorithm for the case of the basic DL $\mathcal{ALC}$ and prove that it is a decision procedure (i.e., that it is sound, complete and terminating);
- show how the algorithm can be extended to deal with some of the extensions described in Section 2.5, and how the proofs of soundness, completeness and termination can be adapted accordingly; and

[1] By "expressive" we mean here DLs that require some form of reasoning by case, for example to handle disjunction.

$$
\begin{aligned}
\mathcal{T}_1 = \quad & \{\mathsf{Course} \equiv \mathsf{UGC} \sqcup \mathsf{PGC}, \\
& \mathsf{PGC} \sqsubseteq \neg\mathsf{UGC}, \\
& \mathsf{Professor} \sqsubseteq \mathsf{Teacher} \sqcap \exists \mathit{teaches}.\mathsf{PGC} \ \} \\
\mathcal{A}_1 = \quad & \{\mathsf{Betty} : \mathsf{Professor}, \\
& \mathsf{Hugo} : \mathsf{Student}, \\
& \mathsf{CS600} : \mathsf{Course}, \\
& (\mathsf{Betty}, \mathsf{CS600}) : \mathit{teaches}, \\
& (\mathsf{Hugo}, \mathsf{CS600}) : \mathit{attends} \ \}
\end{aligned}
$$

Fig. 4.1. Example TBox and ABox.

- briefly review some of the techniques that are used in order to improve the performance of tableau-based reasoners in practice.

4.1 Tableau basics

We recall from Definition 2.14 that a knowledge base $\mathcal{K}$ is consistent if there exists some model $\mathcal{I}$ of $\mathcal{K}$. For example, given the knowledge base $\mathcal{K}_1 = (\mathcal{T}_1, \mathcal{A}_1)$ with $\mathcal{T}_1$ and $\mathcal{A}_1$ defined as in Figure 4.1, it is easy to see that the following interpretation $\mathcal{I}_1$ is a model of $\mathcal{K}_1$ (to really *see* this, we cordially invite the reader to draw the model below following the example given in Figure 2.2):

$$
\begin{aligned}
\Delta^{\mathcal{I}_1} &= \{b, h, c\}, & \mathsf{Betty}^{\mathcal{I}_1} &= b, \\
\mathsf{Hugo}^{\mathcal{I}_1} &= h, & \mathsf{CS600}^{\mathcal{I}_1} &= c, \\
\mathsf{PGC}^{\mathcal{I}_1} &= \{c\}, & \mathsf{UGC}^{\mathcal{I}_1} &= \emptyset, \\
\mathsf{Teacher}^{\mathcal{I}_1} &= \{b\}, & \mathsf{Professor}^{\mathcal{I}_1} &= \{b\}, \\
\mathsf{Student}^{\mathcal{I}_1} &= \{h\}, & \mathsf{Course}^{\mathcal{I}_1} &= \{c\}, \\
\mathit{teaches}^{\mathcal{I}_1} &= \{(b, c)\}, & \mathit{attends}^{\mathcal{I}_1} &= \{(h, c)\}.
\end{aligned}
$$

The existence of $\mathcal{I}_1$ proves that $\mathcal{K}_1$ is consistent; we say that such a model is a *witness* of the consistency of $\mathcal{K}_1$.

The idea behind tableau-based techniques is to try to prove the consistency of a knowledge base $\mathcal{K} = (\mathcal{A}, \mathcal{T})$ by demonstrating the existence of a suitable witness, i.e., an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{K}$. They do this constructively, starting from $\mathcal{A}$ and extending it as needed to explicate constraints implied by the semantics of concepts and axioms in $\mathcal{A}$ and $\mathcal{T}$. This results either in the construction of (an ABox representation of) a witness,[2] or in the discovery of obvious contradictions that

[2] For convenience and brevity, we will sometimes conflate the notions of a witness and the ABox representation of a witness – when our intended meaning is obvious from the context.

prove that no such witness can exist, and thus that $\mathcal{K}$ is not consistent. Note that, in contrast, the consequence-based techniques to be described in Chapter 6 prove that a subsumption follows from a knowledge base by deriving new GCIs (consequences) from the given ones.

The tree model property (see Section 3.5), or some generalisation of it, is critical to the effectiveness and correctness of tableau-based techniques. On the one hand, an algorithm can restrict itself to constructing tree-like witnesses; this is critical for effectiveness, as it greatly reduces the number of possible witnesses that need to be considered, and for completeness, as the non-existence of a tree-like witness is sufficient to prove the non-existence of *any* witness. On the other hand, the structure of tree-like witnesses makes it relatively easy to identify when the construction of some branch of the tree has become repetitive; this is critical for termination, as we can halt the construction of such branches, and for soundness, as we can show that such partially constructed witnesses imply the existence of a complete (but possibly infinite) witness.

4.2 A tableau algorithm for $\mathcal{ALC}$

In this section we will present an algorithm that takes as input an $\mathcal{ALC}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and returns either "consistent" or "inconsistent". We will show that the algorithm terminates, and that it returns "consistent" if and only if $\mathcal{K}$ is consistent; i.e., that it is a decision procedure for $\mathcal{ALC}$ knowledge base consistency.

We will do this in three stages: first, we will present an algorithm for deciding $\mathcal{ALC}$ *ABox consistency*; second, we will show how this algorithm can be extended to one deciding $\mathcal{ALC}$ *knowledge base consistency* in the case where $\mathcal{T}$ is acyclic; and third, we will show how this algorithm can further be extended to deal with the case where $\mathcal{T}$ is an arbitrary TBox.

In the following, unless stated to the contrary, we will assume that A, B, C and D are concepts, that r and s are roles, and that a, b, c and d are individual names. To simplify the presentation, and without loss of generality, we will assume that all concepts occurring in $\mathcal{T}$ or $\mathcal{A}$ are in *negation normal form* (NNF), i.e., that negation is applied only to concept names, that $\mathcal{A}$ is non-empty and that every individual name occurring in $\mathcal{A}$ occurs in at least one assertion of the form $a : C$; we will call such an ABox *normalised*. An arbitrary $\mathcal{ALC}$ concept can be transformed into an equivalent one in NNF by pushing negations inwards using a combination of de Morgan's laws and the duality between exis-

tential and universal restrictions, as well as eliminating double negation (see Lemma 2.3):

$$\begin{array}{rcl@{\qquad}rcl} \neg(C \sqcap D) & \equiv & \neg C \sqcup \neg D, & \neg(C \sqcup D) & \equiv & \neg C \sqcap \neg D, \\ \neg\exists r.C & \equiv & \forall r.\neg C, & \neg\forall r.C & \equiv & \exists r.\neg C, \\ \neg\neg C & \equiv & C. & & & \end{array}$$

For a concept C, we will use $\dot{\neg}C$ to denote the NNF of $\neg C$. Finally, for any individual name a occurring in $\mathcal{A}$ we can add to $\mathcal{A}$ a vacuous assertion $a:\top$, and if $\mathcal{A}$ is empty we can add the assertion $a:\top$ for some new individual name a.

It will be useful to extend the definition of subconcept (see Section 3.4) to ABoxes and to knowledge bases in the obvious way:

$$\mathsf{sub}(\mathcal{A}) = \bigcup_{a:C \in \mathcal{A}} \mathsf{sub}(C),$$

and, for $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, $\mathsf{sub}(\mathcal{K}) = \mathsf{sub}(\mathcal{T}) \cup \mathsf{sub}(\mathcal{A})$.

4.2.1 ABox consistency

We will first describe an algorithm for deciding ABox consistency, i.e., for the case where $\mathcal{K} = (\emptyset, \mathcal{A})$. This algorithm is very simple because, when the TBox is empty, the expansion rules only need to explicate the semantics of the concepts occurring in concept assertions in $\mathcal{A}$. Moreover, because these rules syntactically decompose concepts, the algorithm naturally terminates when all concepts have been fully decomposed.

As we saw in Section 3.5, $\mathcal{ALC}$ has the tree model property; i.e., every satisfiable concept has a tree model. However, since $\mathcal{A}$ might include individual names connected via arbitrary role assertions, this must be generalised to a *forest model property* for $\mathcal{ALC}$ knowledge bases: if $\mathcal{K}$ is consistent, then it has a model that consists of one or more disjoint trees, where the root of each tree interprets some individual name in $\mathcal{A}$, and where the roots are arbitrarily connected by edges. The algorithm will try to construct a forest-shaped ABox. It will do this by applying *expansion rules* so as to extend $\mathcal{A}$ until it is *complete*. In a complete ABox, consistency can be checked by looking for obvious contradictions (clashes).

Definition 4.1 (Complete and clash-free ABox)**.** An ABox $\mathcal{A}$ contains a *clash* if, for some individual name a, and for some concept C, $\{a:C, a:\neg C\} \subseteq \mathcal{A}$; it is *clash-free* if it does not contain a clash. $\mathcal{A}$

$\sqcap$-rule: if 1. $a : C \sqcap D \in \mathcal{A}$, and
2. $\{a : C, a : D\} \not\subseteq \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : C, a : D\}$

$\sqcup$-rule: if 1. $a : C \sqcup D \in \mathcal{A}$, and
2. $\{a : C, a : D\} \cap \mathcal{A} = \emptyset$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : X\}$ for some $X \in \{C, D\}$

$\exists$-rule: if 1. $a : \exists r.C \in \mathcal{A}$, and
2. there is no b such that $\{(a, b) : r, b : C\} \subseteq \mathcal{A}$,
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{(a, d) : r, d : C\}$, where d is new in $\mathcal{A}$

$\forall$-rule: if 1. $\{a : \forall r.C, (a, b) : r\} \subseteq \mathcal{A}$, and
2. $b : C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{b : C\}$

Fig. 4.2. The syntax expansion rules for $\mathcal{ALC}$ ABox consistency.

is *complete* if it contains a clash, or if none of the expansion rules is applicable.

The definition of a complete ABox refers to the applicability of expansion rules. In the context of this section, these are the expansion rules of Figure 4.2. In later sections we will modify and/or extend the set of rules to deal with TBoxes and additional concept and role constructors. The notion of a complete ABox will remain the same, relative to the modified/extended set of rules, but the notion of a clash may need to be extended to deal with additional constructors.

We are now ready to define an algorithm consistent for deciding the consistency of a normalised $\mathcal{ALC}$ ABox $\mathcal{A}$. Nondeterministic algorithms are often used for this purpose, and have the advantage of being very simple and elegant: a typical definition simply says that $\mathcal{A}$ is consistent if and only if the rules can be applied to it in such a way as to construct a complete and clash-free ABox. However, such an algorithm cannot be directly implemented, and it conflates *relevant* (sometimes called *don't know*) nondeterminism, where different choices may affect the outcome of the algorithm, with *irrelevant* (sometimes called *don't care*) nondeterminism, where the choices made do not affect the outcome. We will instead define a deterministic algorithm that uses search to explore only relevant nondeterministic choices.

In the case of $\mathcal{ALC}$, there is only one such choice: the choice associated with the $\sqcup$-rule. Unlike the other rules, where application of the rule leads deterministically to a unique expanded ABox, the $\sqcup$-rule can be applied in different ways, and applying the rule in the

"wrong" way can change $\mathcal{A}$ from being consistent to being inconsistent. For example, $\{a:\neg D, a:C \sqcup D\}$ is clearly consistent, and can be expanded using the $\sqcup$-rule into the complete and clash-free ABox $\{a:\neg D, a:C \sqcup D, a:C\}$; however, applying the $\sqcup$-rule in the other way would give $\{a:\neg D, a:C \sqcup D, a:D\}$, which is clearly inconsistent – in fact it already contains a clash. In a nondeterministic algorithm we simply say that $\mathcal{A}$ is consistent if we can choose *some* way of applying the $\sqcup$-rule that results in a consistent ABox; in our deterministic algorithm we will (recursively) check the consistency of the ABoxes resulting from *each* possible way of applying the $\sqcup$-rule, and we will say that $\mathcal{A}$ is consistent if *any* of these ABoxes is consistent.

Note that our algorithm does not search different possible orders of rule applications. This is because the order of rule applications does not affect consistency, although it can (dramatically) affect the efficiency of the algorithm. For this reason, an implementation typically chooses the order of rule applications using heuristics that aim to reduce the size of the search space; e.g., they may choose to apply the $\sqcup$-rule only if no other rule is applicable (see Section 4.4). Moreover, we can freely choose the order in which to explore the different expansion choices offered by the $\sqcup$-rule – if any choice leads to a consistent ABox, then our algorithm will (eventually) find it – and in practice this order may also be heuristically determined.

To facilitate the description of our deterministic algorithm we introduce a function exp that takes as input a normalised and clash-free $\mathcal{ALC}$ ABox $\mathcal{A}$, a rule R and an assertion or pair of assertions α such that R is applicable to α in $\mathcal{A}$; it returns a set $\mathsf{exp}(\mathcal{A}, R, \alpha)$ containing each of the ABoxes that can result from applying R to α in $\mathcal{A}$. For example,

- $\mathsf{exp}(\{a:\neg D, a:C \sqcup D\}, \sqcup\text{-rule}, a:C \sqcup D)$ returns a set containing two ABoxes: $\{a:\neg D, a:C \sqcup D, a:C\}$ and $\{a:\neg D, a:C \sqcup D, a:D\}$;
- $\mathsf{exp}(\{b:\neg D, a:\forall r.D, (a,b):r\}, \forall\text{-rule}, (a:\forall r.D, (a,b):r))$ returns a singleton set consisting of the ABox $\{b:\neg D, a:\forall r.D, (a,b):r, b:D\}$.

For deterministic rules exp returns singleton sets, whereas for nondeterministic rules it returns sets of cardinality greater than one. In the case of $\mathcal{ALC}$, the $\sqcup$-rule is the only such nondeterministic rule, always returning sets of cardinality two, but, as we will see in Section 4.3, extending the algorithm to deal with additional constructors may necessitate the introduction of additional nondeterministic rules, which may also return sets of larger cardinality.

Algorithm consistent()
Input: a normalised $\mathcal{ALC}$ ABox $\mathcal{A}$
if expand($\mathcal{A}$) $\neq \emptyset$ **then**
  return "consistent"
else
  return "inconsistent"

Algorithm expand()
Input: a normalised $\mathcal{ALC}$ ABox $\mathcal{A}$
if $\mathcal{A}$ is not complete **then**
  select a rule R that is applicable to $\mathcal{A}$ and an assertion or pair of assertions α in $\mathcal{A}$ to which R is applicable
  if there is $\mathcal{A}' \in \mathsf{exp}(\mathcal{A}, R, \alpha)$ with expand($\mathcal{A}'$) $\neq \emptyset$ **then**
    return expand($\mathcal{A}'$)
  else
    return $\emptyset$
else
  if $\mathcal{A}$ contains a clash **then**
    return $\emptyset$
  else
    return $\mathcal{A}$

Fig. 4.3. The tableau algorithm consistent for $\mathcal{ALC}$ ABox consistency and the ABox expansion algorithm expand.

Definition 4.2 (Algorithm for $\mathcal{ALC}$ ABox consistency)**.** The algorithm consistent for $\mathcal{ALC}$ ABox consistency takes as input a normalised $\mathcal{ALC}$ ABox $\mathcal{A}$ and uses the algorithm expand to apply the rules from Figure 4.2 to $\mathcal{A}$; both algorithms are given in Figure 4.3.

Before discussing the properties of the algorithm consistent in detail, and proving that it is in fact a decision procedure for $\mathcal{ALC}$ ABox consistency, we will use the following example ABox to illustrate some important features of consistent:

$$\begin{aligned}\mathcal{A}_{ex} = \{&a : A \sqcap \exists s.F, && (a,b) : s,\\ &a : \forall s.(\neg F \sqcup \neg B), && (a,c) : r,\\ &b : B, && c : C \sqcap \exists s.D\}.\end{aligned}$$

Note that $\mathcal{A}_{ex}$ is already normalised, and so the algorithm can be applied to it.

First, we note that a precondition for the application of each rule to an ABox $\mathcal{A}$ is the presence in $\mathcal{A}$ of a concept assertion $e : E$, where E is

of the relevant type ($\sqcap$, $\sqcup$, $\exists$ or $\forall$), and in each case the rule only adds concept assertions of the form $e : E'$ or $f : E'$, where E' is a subconcept of E and f an individual name such that $(e, f) : t \in \mathcal{A}$ for a role t.

For example, applying the $\sqcap$-rule to the first assertion in $\mathcal{A}_{ex}$ yields $\mathcal{A} = \mathcal{A}_{ex} \cup \{a : A,\ a : \exists s.F\}$. Note that we could instead have chosen to apply the $\sqcap$-rule to $c : C \sqcap \exists s.D$; however, as mentioned above, such choices do not affect the eventual outcome (i.e., whether the algorithm returns "consistent" or "inconsistent"), as rules remain applicable until their consequents have been satisfied (in this case, until both $c : C$ and $c : \exists s.D$ have been added).

Second, new individual names are introduced by the $\exists$-rule (and by no other rule), and such individual names are connected to an existing individual name by a single role assertion. Hence such individual names form trees whose roots are the individual names that occur in the input ABox; the resulting ABox can thus be said to form a forest.

In our example, applying the $\exists$-rule to $a : \exists s.F$ adds the assertions $(a, x) : s,\ x : F$, where x is a new individual name (i.e., different from the individual names a, b, c already occurring in the ABox). In subsequent steps, we can apply the $\forall$-rule to $a : \forall s.(\neg F \sqcup \neg B)$ together with $(a, b) : s$ and $(a, x) : s$. For the first role assertion, the rule adds $b : \neg F \sqcup \neg B$; for the second, it adds $x : \neg F \sqcup \neg B$.

Third, as discussed above, the $\sqcup$-rule is nondeterministic, and we have to explore all possible ways of applying it until we find one that leads to the construction of a complete and clash-free ABox, or determine that none of them does. In our example, we can apply the $\sqcup$-rule to $x : \neg F \sqcup \neg B$ in the current ABox $\mathcal{A}$, and in this case $\mathsf{exp}(\mathcal{A}, \sqcup\text{-rule}, x : \neg F \sqcup \neg B) = \{\mathcal{A} \cup \{x : \neg F\}, \mathcal{A} \cup \{x : \neg B\}\}$. If we first try $\mathcal{A}' = \mathcal{A} \cup \{x : \neg F\}$, then we will find that $\mathsf{expand}(\mathcal{A}') = \emptyset$ ($\mathcal{A}'$ already contains a clash); we will then try $\mathcal{A}' = \mathcal{A} \cup \{x : \neg B\}$. Similarly, we can apply the $\sqcup$-rule to $b : \neg F \sqcup \neg B \in \mathcal{A}$; in this case, if we first try $\mathcal{A}' = \mathcal{A} \cup \{b : \neg F\}$, then we will find that it is consistent and we won't try $\mathcal{A}' = \mathcal{A} \cup \{b : \neg B\}$.

We can now finish the example, assuming that we have so far extended

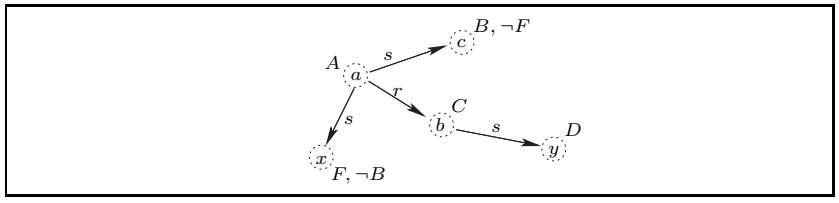

Fig. 4.4. A graphical representation of the complete and clash-free ABox generated for $\mathcal{A}_{ex}$ by the tableau algorithm in Figure 4.3.

the ABox $\mathcal{A}_{ex}$ to

$$\begin{aligned}\mathcal{A} = \{&a : A \sqcap \exists s.F, && (a,b) : s,\\ &a : \forall s.(\neg F \sqcup \neg B), && (a,c) : r,\\ &b : B, && c : C \sqcap \exists s.D,\\ &a : A, && a : \exists s.F,\\ &x : F, && (a,x) : s,\\ &b : \neg F \sqcup \neg B, && x : \neg F \sqcup \neg B,\\ &b : \neg F, && x : \neg B\}.\end{aligned}$$

Only the $\sqcap$-rule is applicable to $c : C \sqcap \exists s.D \in \mathcal{A}$, and its application adds $c : C$ and $c : \exists s.D$ to $\mathcal{A}$. Now the $\exists$-rule is applicable to $c : \exists s.D \in \mathcal{A}$, and its application adds $(c, y) : s$ and $y : D$ to $\mathcal{A}$. We have now constructed a complete and clash-free ABox $\mathcal{A}$, and thus the tableau algorithm returns "consistent". A graphical representation of $\mathcal{A}$ can be found in Figure 4.4, where we leave out all complex concept expressions.

Before we analyse the algorithm in detail, we introduce some useful notation: as individual names introduced by the $\exists$-rule form part of a tree, we will call them *tree individuals*, and we will call the individual names that occur in the input ABox *root individuals*. It will sometimes be convenient to refer to the *predecessor* and *successor* of an individual name, defining them in the obvious way: if the $\exists$-rule adds a tree individual b and a role assertion of the form $(a, b) : r$, then b is a (r-) successor of a and a is a predecessor of b. We will use *ancestor* and *descendant* for the transitive closure of predecessor and successor, respectively; i.e., if a is the predecessor (successor) of b, then a is also an ancestor (descendant) of b, and if a is an ancestor (descendant) of b' and b' is an ancestor (descendant) of b, then a is an ancestor (descendant) of b. Note that root individuals may have successors (and hence descendants), but they have no predecessor (and hence no ancestors).

We will denote by $\mathsf{con}_{\mathcal{A}}(a)$ the set of concepts C in concept assertions

of the form $a : C$, i.e.,

$$\mathsf{con}_{\mathcal{A}}(a) = \{C \mid a : C \in \mathcal{A}\}.$$

The following lemma is an immediate consequence of the fact that $|\mathsf{sub}(C)| \leq \mathsf{size}(C)$, which was shown in Lemma 3.11.

Lemma 4.3. *For each $\mathcal{ALC}$ ABox $\mathcal{A}$, we have that* $|\mathsf{sub}(\mathcal{A})| \leq \sum_{a:C \in \mathcal{A}} \mathsf{size}(C)$.

As a consequence of Lemma 4.3, we can say that the cardinality of $\mathsf{sub}(\mathcal{A})$ is linear in the size of $\mathcal{A}$.

Please note that, in our example, the tree individuals form rather trivial trees; we invite the reader to run the algorithm again on the given ABox extended with $b : \exists r.A \sqcap \exists r.B \sqcap \forall r.(\exists s.A \sqcap \exists s.B)$ to see a less trivial tree of six freshly introduced tree individuals.

We will now prove that, for any $\mathcal{ALC}$ ABox $\mathcal{A}$, the algorithm terminates, returns "consistent" only if $\mathcal{A}$ is consistent (i.e., it is sound), and returns "consistent" whenever $\mathcal{A}$ is consistent (i.e., it is complete).

Lemma 4.4 (Termination)**.** *For each $\mathcal{ALC}$ ABox $\mathcal{A}$,* $\mathsf{consistent}(\mathcal{A})$ *terminates.*

Proof. Let $m = |\mathsf{sub}(\mathcal{A})|$. Termination is a consequence of the following properties of the expansion rules:

(i) The expansion rules never remove an assertion from $\mathcal{A}$, and each rule application adds a new assertion of the form $a : C$, for some individual name a and some concept $C \in \mathsf{sub}(\mathcal{A})$. Moreover, we saw in Lemmas 3.11 and 4.3 that the size of $\mathsf{sub}(\mathcal{A})$ is bounded by the size of $\mathcal{A}$, and thus there can be at most m rule applications adding a concept assertion of the form $a : C$ for any individual name a, and $|\mathsf{con}_{\mathcal{A}}(a)| \leq m$.

(ii) A new individual name is added to $\mathcal{A}$ only when the $\exists$-rule is applied to an assertion of the form $a : C$ with C an existential restriction (a concept of the form $\exists r.D$), and for any individual name each such assertion can trigger the addition of at most one new individual name. As there can be no more than m different existential restrictions in $\mathcal{A}$, a given individual name can cause the addition of at most m new individual names, and the out-degree of each tree in the forest-shaped ABox is thus bounded by m.

(iii) The $\exists$- and $\forall$-rules are triggered by assertions of the form $a : \exists r.C$ and $a : \forall r.C$, respectively, and they only add concept assertions of the form $b : C$, where b is a successor of a; in either case, C is a strict subdescription of the concept $\exists r.C$ or $\forall r.C$ in the assertion to which the rule was applied, and it is clearly strictly smaller than these concepts. Further rule applications may be triggered by the presence of $b : C$ in $\mathcal{A}$, adding additional concept assertions $b : D$, but then D is a subdescription of C that is smaller than C, etc. Consequently, $\mathsf{sub}(\mathsf{con}_{\mathcal{A}}(b)) \subseteq \mathsf{sub}(\mathsf{con}_{\mathcal{A}}(a))$ and the size of the largest concept in $\mathsf{con}_{\mathcal{A}}(b)$ is smaller than the size of the largest concept in $\mathsf{con}_{\mathcal{A}}(a)$. The second fact shows that the inclusion stated by the first fact is actually strict; i.e., for any tree individual b whose predecessor is a, $\mathsf{sub}(\mathsf{con}_{\mathcal{A}}(b)) \subsetneq \mathsf{sub}(\mathsf{con}_{\mathcal{A}}(a))$. Consequently, the depth of each tree in the forest-shaped ABox is bounded by m.

These properties ensure that there is a bound on the size of the ABox that can be constructed via rule applications, and thus a bound on the number of recursive applications of expand. □

Lemma 4.5 (Soundness)**.** *If* $\mathsf{consistent}(\mathcal{A})$ *returns "consistent", then* $\mathcal{A}$ *is consistent.*

Proof. Let $\mathcal{A}'$ be the set returned by $\mathsf{expand}(\mathcal{A})$. Since the algorithm returns "consistent", $\mathcal{A}'$ is a complete and clash-free ABox.

The proof then follows rather easily from the very close correspondence between $\mathcal{A}'$ and an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ that is a model of $\mathcal{A}'$, i.e., that satisfies each assertion in $\mathcal{A}'$. Given that the expansion rules never delete assertions, we have that $\mathcal{A} \subseteq \mathcal{A}'$, so $\mathcal{I}$ is also a model of $\mathcal{A}$, and is a witness to the consistency of $\mathcal{A}$. We use $\mathcal{A}'$ to construct a suitable interpretation $\mathcal{I}$ as follows:

$$\begin{aligned}
\Delta^{\mathcal{I}} &= \{a \mid a : C \in \mathcal{A}'\},\\
a^{\mathcal{I}} &= a \text{ for each individual name } a \text{ occurring in } \mathcal{A}',\\
A^{\mathcal{I}} &= \{a \mid A \in \mathsf{con}_{\mathcal{A}'}(a)\} \text{ for each concept name } A \text{ in } \mathsf{sub}(\mathcal{A}'),\\
r^{\mathcal{I}} &= \{(a, b) \mid (a, b) : r \in \mathcal{A}'\} \text{ for each role } r \text{ occurring in } \mathcal{A}'.
\end{aligned}$$

Note that each individual name a that occurs in $\mathcal{A}'$ occurs in at least one concept assertion: for root individuals this follows from our assumptions on the structure of $\mathcal{A}$, and for tree individuals this follows from the definition of the $\exists$-rule. It is easy to see that $\mathcal{I}$ is an interpretation: by assumption, $\mathcal{A}$ contains at least one assertion of the form $a : C$, so $\Delta^{\mathcal{I}}$

is non-empty, and by construction $\cdot^{\mathcal{I}}$ maps every individual name in $\mathcal{A}'$ to an element of $\Delta^{\mathcal{I}}$, every concept name $A \in \mathsf{sub}(\mathcal{A}')$ to a subset of $\Delta^{\mathcal{I}}$, and every role r occurring in $\mathcal{A}'$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. From Definition 2.6, $\mathcal{I}$ is a model of $\mathcal{A}'$ if it satisfies each concept and role assertion in $\mathcal{A}'$. The construction of $\mathcal{I}$ means that it trivially satisfies all role assertions in $\mathcal{A}'$. By induction on the structure of concepts, we show the following property (P1):

$$\text{if } a : C \in \mathcal{A}', \text{ then } a^{\mathcal{I}} \in C^{\mathcal{I}}. \tag{P1}$$

Induction Basis C is a concept name: by definition of $\mathcal{I}$, if $a : C \in \mathcal{A}'$, then $a^{\mathcal{I}} \in C^{\mathcal{I}}$ as required.

Induction Steps

- $C = \neg D$: since $\mathcal{A}'$ is clash-free, $a : \neg D \in \mathcal{A}'$ implies that $a : D \notin \mathcal{A}'$. Since all concepts in $\mathcal{A}$ are in NNF, D is a concept name. By definition of $\mathcal{I}$, $a^{\mathcal{I}} \notin D^{\mathcal{I}}$, which implies $a^{\mathcal{I}} \in \Delta^{\mathcal{I}} \setminus D^{\mathcal{I}} = C^{\mathcal{I}}$ as required.
- $C = D \sqcup E$: if $a : D \sqcup E \in \mathcal{A}'$, then completeness of $\mathcal{A}'$ implies that $\{a : D, a : E\} \cap \mathcal{A}' \neq \emptyset$ (otherwise the $\sqcup$-rule would be applicable). Thus $a^{\mathcal{I}} \in D^{\mathcal{I}}$ or $a^{\mathcal{I}} \in E^{\mathcal{I}}$ by induction, and hence $a^{\mathcal{I}} \in D^{\mathcal{I}} \cup E^{\mathcal{I}} = (D \sqcup E)^{\mathcal{I}}$ by the semantics of $\sqcup$.
- $C = D \sqcap E$: this case is analogous to but easier than the previous one and is left to the reader as a useful exercise.
- $C = \forall r.D$: let $a : \forall r.D \in \mathcal{A}'$ and consider b with $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$. For $a^{\mathcal{I}}$ to be in $(\forall r.D)^{\mathcal{I}}$, we need to ensure that $b^{\mathcal{I}} \in D^{\mathcal{I}}$. By definition of $\mathcal{I}$, $(a, b) : r \in \mathcal{A}'$. Since $\mathcal{A}'$ is complete and $a : \forall r.D \in \mathcal{A}'$, we have that $b : D \in \mathcal{A}'$ (otherwise the $\forall$-rule would be applicable). By induction, $b^{\mathcal{I}} \in D^{\mathcal{I}}$, and since the above holds for all b with $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$, we have that $a^{\mathcal{I}} \in (\forall r.D)^{\mathcal{I}}$ by the semantics of $\forall$.
- $C = \exists r.D$: again, this case is analogous to and easier than the previous one and is left to the reader as a useful exercise.

As a consequence, $\mathcal{I}$ satisfies all concept assertions in $\mathcal{A}'$ and thus in $\mathcal{A}$, and it satisfies all role assertions in $\mathcal{A}'$ and thus in $\mathcal{A}$ by definition. Hence $\mathcal{A}$ has a model and thus is consistent. □

Lemma 4.6 (Completeness)**.** *If $\mathcal{A}$ is consistent, then* $\mathsf{consistent}(\mathcal{A})$ *returns "consistent".*

Proof. Let $\mathcal{A}$ be consistent, and consider a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{A}$. Since $\mathcal{A}$ is consistent, it cannot contain a clash.

If $\mathcal{A}$ is complete – since it does not contain a clash – expand simply returns $\mathcal{A}$ and consistent returns "consistent". If $\mathcal{A}$ is not complete, then expand calls itself recursively until $\mathcal{A}$ is complete; each call selects a rule and applies it. We will show that rule application preserves consistency by a case analysis according to the type of rule:

- The $\sqcup$-rule: If $a : C \sqcup D \in \mathcal{A}$, then $a^{\mathcal{I}} \in (C \sqcup D)^{\mathcal{I}}$ and Definition 2.2 implies that either $a^{\mathcal{I}} \in C^{\mathcal{I}}$ or $a^{\mathcal{I}} \in D^{\mathcal{I}}$. Therefore, at least one of the ABoxes $\mathcal{A}' \in \mathsf{exp}(\mathcal{A}, \sqcup\text{-rule}, a : C \sqcup D)$ is consistent. Thus, one of the calls of expand is applied to a consistent ABox.
- The $\sqcap$-rule: If $a : C \sqcap D \in \mathcal{A}$, then $a^{\mathcal{I}} \in (C \sqcap D)^{\mathcal{I}}$ and Definition 2.2 implies that both $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and $a^{\mathcal{I}} \in D^{\mathcal{I}}$. Therefore, $\mathcal{I}$ is still a model of $\mathcal{A} \cup \{a : C, a : D\}$, so $\mathcal{A}$ is still consistent after the rule is applied.
- The $\exists$-rule: If $a : \exists r.C \in \mathcal{A}$, then $a^{\mathcal{I}} \in (\exists r.C)^{\mathcal{I}}$ and Definition 2.2 implies that there is some $x \in \Delta^{\mathcal{I}}$ such that $(a^{\mathcal{I}}, x) \in r^{\mathcal{I}}$ and $x \in C^{\mathcal{I}}$. Therefore, there is a model $\mathcal{I}'$ of $\mathcal{A}$ such that, for some new individual name d, $d^{\mathcal{I}'} = x$, and that is otherwise identical to $\mathcal{I}$. This model $\mathcal{I}'$ is still a model of $\mathcal{A} \cup \{(a, d) : r, d : C\}$, so $\mathcal{A}$ is still consistent after the rule is applied.
- The $\forall$-rule: If $\{a : \forall r.C, (a, b) : r\} \subseteq \mathcal{A}$, then $a^{\mathcal{I}} \in (\forall r.C)^{\mathcal{I}}$, $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$, and Definition 2.2 implies that $b^{\mathcal{I}} \in C^{\mathcal{I}}$. Therefore, $\mathcal{I}$ is still a model of $\mathcal{A} \cup \{b : C\}$, so $\mathcal{A}$ is still consistent after the rule is applied.

□

Theorem 4.7. *The tableau algorithm presented in Definition 4.2 is a decision procedure for the consistency of $\mathcal{ALC}$ ABoxes.*

Proof. That the algorithm is a decision procedure for normalised $\mathcal{ALC}$ ABoxes follows from Lemmas 4.4, 4.5 and 4.6; and as we showed at the beginning of this subsection, an arbitrary $\mathcal{ALC}$ ABox can be transformed into an equivalent normalised ABox. □

A few comments on the complexity of the algorithm are appropriate at this point. As we will see in Section 5.1, the complexity of the $\mathcal{ALC}$ ABox consistency problem is PSPACE-complete; however, the fully expanded ABox may be exponentially larger than the input ABox, and applications of nondeterministic rules may lead to the exploration of exponentially many different ABoxes, so the algorithm as presented above requires exponential time and space. The algorithm can, however, easily be adapted to use only polynomial space via a so-called trace technique [SS91]. This relies firstly on the fact that individual names introduced

$\sqsubseteq$-rule: if 1. $a : A \in \mathcal{A}$, $A \sqsubseteq C \in \mathcal{T}$, and
2. $a : C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : C\}$

$\equiv_1$-rule: if 1. $a : A \in \mathcal{A}$, $A \equiv C \in \mathcal{T}$, and
2. $a : C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : C\}$

$\equiv_2$-rule: if 1. $a : \neg A \in \mathcal{A}$, $A \equiv C \in \mathcal{T}$, and
2. $a : \dot{\neg} C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : \dot{\neg} C\}$

Fig. 4.5. The axiom unfolding rules for $\mathcal{ALC}$.

by the $\exists$-rule form part of a tree, and secondly on the fact that the order of rule applications can be chosen arbitrarily. Exploiting the second property, we can exhaustively apply the $\sqcap$, $\sqcup$ and $\forall$-rules to existing individual names before considering the $\exists$-rule. We can then construct the tree parts of the forest model one branch at a time, reusing space once we have completed the construction of a given branch. For example, if $\mathcal{A} = \{a : \exists r.C, a : \exists r.D, a : \forall r.A\}$, then we can apply the $\exists$- and $\forall$-rules to introduce new individual names b and c with $\{b : C, b : A\} \subseteq \mathcal{A}$ and $\{c : D, c : A\} \subseteq \mathcal{A}$. The consistency of $\{b : C, b : A\}$ and $\{c : D, c : A\}$ can then be treated as independent sub-problems, with the space used to solve each of them being subsequently reused.

4.2.2 Acyclic knowledge base consistency

We can use the algorithm from Definition 4.2 to decide the consistency of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ where $\mathcal{T}$ is acyclic as per Definition 2.9, i.e., where all axioms in $\mathcal{T}$ are of the form $A \equiv C$ or $A \sqsubseteq C$ for A a concept name and C a possibly compound $\mathcal{ALC}$ concept that does not use A either directly or indirectly. In such cases, we can unfold $\mathcal{T}$ into $\mathcal{A}$ to give $\mathcal{A}'$ as per Definition 2.11, and from Lemma 2.12 it follows that $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent if and only if $\mathcal{K}' = (\emptyset, \mathcal{A}')$ is consistent.

As shown in Example 2.13, this *eager* unfolding procedure could lead to an exponential increase in the size of the ABox. This can be avoided by unfolding definitions only as required by the progress of the algorithm, so-called *lazy* unfolding. Suitable lazy unfolding rules are presented in Figure 4.5; when used in conjunction with the syntax expansion rules from Figure 4.2 the resulting algorithm is a decision procedure for acyclic $\mathcal{ALC}$ knowledge bases. Rather than adapting the proofs from the previous section to this case, we are going to generalise both the approach

and the proofs to general TBoxes, i.e., those possibly involving cyclic dependencies between concept names such as $A \sqsubseteq \exists r.A$ and axioms with complex concepts on the left-hand side such as $\exists r.B \sqsubseteq \forall s.E$. For the interested reader who wants to adapt the proofs to the acyclic TBox case, termination is rather straightforward, as is completeness, with the main changes to be done in the soundness proof. To construct a model from a complete and clash-free ABox $\mathcal{A}'$, one can use the same definitions as in the proof of Lemma 4.5 to obtain an interpretation $\mathcal{J}$ of all role names and of the concept names that do not have definitions in $\mathcal{T}$. This can then be extended to a model $\mathcal{I}$ of $\mathcal{T}$ by interpreting defined concepts A with definition $A \equiv C \in \mathcal{T}$ in the same way as $\mathcal{J}$ interprets C (see the proof of Lemma 2.10). It remains to show that $\mathcal{I}$ is also a model of $\mathcal{A}'$, where the main problem is showing Property (P1) by induction. For this, one needs to define a well-founded order in which any concept is larger than its strict subconcepts *and* A is larger than C if $A \equiv C \in \mathcal{T}$.

4.2.3 General knowledge base consistency

Next, we present a tableau algorithm that deals with "full" $\mathcal{ALC}$ knowledge bases, i.e., an ABox and a general TBox.

As a consequence of Lemma 2.16, we have the following two equivalences, and we can thus assume without loss of generality that all our TBox axioms are of the form $\top \sqsubseteq E$:

$\mathcal{I}$ satisfies $C \sqsubseteq D$ if and only if $\mathcal{I}$ satisfies $\top \sqsubseteq D \sqcup \neg C$,
$\mathcal{I}$ satisfies $C \equiv D$ if and only if $\mathcal{I}$ satisfies $\top \sqsubseteq (D \sqcup \neg C) \sqcap (C \sqcup \neg D)$.

We will extend our notion of *normalised* to TBoxes and knowledge bases accordingly: we will say that a TBox is normalised if its constituent axioms are all of the form $\top \sqsubseteq E$, where E is in NNF; and that a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is normalised if both $\mathcal{T}$ and $\mathcal{A}$ are normalised.

In order to deal with these GCIs, we extend the tableau algorithm from Section 4.2.1 to use the expansion rules shown in Figure 4.6: they are identical to the ones shown for ABoxes in Figure 4.2 apart from the addition of the $\sqsubseteq$-rule and the third clause in the $\exists$-rule. The former deals with GCIs, the latter ensures termination, and both will be explained later – after we have explained why termination needs to be dealt with explicitly. Regarding the latter, please note that, without the third clause in the $\exists$-rule, the resulting algorithm can still be proven to be sound and complete, but it would no longer be guaranteed to terminate.

$\sqcap$-rule: if 1. $a : C \sqcap D \in \mathcal{A}$, and
2. $\{a : C, a : D\} \not\subseteq \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : C, a : D\}$

$\sqcup$-rule: if 1. $a : C \sqcup D \in \mathcal{A}$, and
2. $\{a : C, a : D\} \cap \mathcal{A} = \emptyset$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : X\}$ for some $X \in \{C, D\}$

$\exists$-rule: if 1. $a : \exists r.C \in \mathcal{A}$,
2. there is no b such that $\{(a, b) : r, b : C\} \subseteq \mathcal{A}$, and
3. a is not blocked,
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{(a, d) : r, d : C\}$, where d is new in $\mathcal{A}$

$\forall$-rule: if 1. $\{a : \forall r.C, (a, b) : r\} \subseteq \mathcal{A}$, and
2. $b : C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{b : C\}$

$\sqsubseteq$-rule: if 1. $a : C \in \mathcal{A}$, $\top \sqsubseteq D \in \mathcal{T}$, and
2. $a : D \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{a : D\}$

Fig. 4.6. The syntax expansion rules for $\mathcal{ALC}$ KB consistency.

The termination problem stems from the fact that a naive combination of the $\exists$- and $\sqsubseteq$-rules could introduce a successor b of a such that $\mathsf{sub}(\mathsf{con}_{\mathcal{A}}(b))$ is no longer a strict subset of $\mathsf{sub}(\mathsf{con}_{\mathcal{A}}(a))$, and so the depth of trees in the forest-shaped ABox is no longer naturally bounded; as an example, consider the knowledge base $(\{A \sqsubseteq \exists r.A\},\ \{a : A\})$ or its normalised equivalent $(\{\top \sqsubseteq \neg A \sqcup \exists r.A\},\ \{a : A\})$.

For $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and any individual name a in the ABox during the run of the tableau algorithm, the set $\mathsf{con}_{\mathcal{A}}(a)$ is contained in $\mathsf{sub}(\mathcal{K}) = \mathsf{sub}(\mathcal{T}) \cup \mathsf{sub}(\mathcal{A})$, and thus any branch in a tree can contain at most $2^{|\,\mathsf{sub}(\mathcal{K})|}$ individual names before it contains two different individual names a and b such that $\mathsf{con}_{\mathcal{A}}(a) = \mathsf{con}_{\mathcal{A}}(b)$. If this situation arises, then the rules could clearly be applied to b so as to (eventually) introduce another individual name b' with $\mathsf{con}_{\mathcal{A}}(b) = \mathsf{con}_{\mathcal{A}}(b')$, and the construction could continue like this indefinitely. However, as we will see later, if the ABox is clash-free, then we can stop our ABox construction and use the regularity of such branches to define a suitable ABox (and hence model) without explicitly introducing b' or further "clones" of a.

To formalise this idea and thus ensure termination,[3] we halt construction of a given branch once it contains two individual names that can be considered "clones" of each other, a technique known as *blocking*.

[3] We could do this simply by imposing the relevant depth limit on the construction, but the approach presented is more efficient.

Definition 4.8 ($\mathcal{ALC}$ blocking)**.** An individual name b in an $\mathcal{ALC}$ ABox $\mathcal{A}$ is *blocked by an individual name* a if

- a is an ancestor of b, and
- $\mathsf{con}_{\mathcal{A}}(a) \supseteq \mathsf{con}_{\mathcal{A}}(b)$.

An individual name b is *blocked in* $\mathcal{A}$ if it is blocked by some individual name a, or if one or more of its ancestors is blocked in $\mathcal{A}$. When it is clear from the context, we may not mention the ABox explicitly; e.g., we may simply say that b is *blocked*.

Please note the following two consequences of this definition: (a) when an individual name is blocked, all of its descendants are also blocked; and (b) since a root individual has no ancestors it can never be blocked.

As mentioned above, blocking guarantees termination without compromising soundness. To prove soundness of our algorithm, we construct, from a complete and clash-free ABox $\mathcal{A}$, a model of the input knowledge base. For this construction, if b is blocked by a in $\mathcal{A}$, then we have two equally valid choices:

(i) we repeat the structure of the section between a and b infinitely often, leading to an infinite tree model, or
(ii) instead of introducing b, the branch loops back to a, leading to a finite model with cycles.

As we saw in Sections 3.4 and 3.5, $\mathcal{ALC}$ TBoxes have both the finite model property and the tree model property. Given a consistent knowledge base $\mathcal{K}$, the first choice above leads to an (infinite) tree model[4] witness (of the consistency of $\mathcal{K}$), whereas the second choice leads to a finite (non-tree) model witness. As mentioned in Section 3.5, there are $\mathcal{ALC}$ concepts that, with respect to a general TBox, do not have a model that is both finite *and* tree-shaped. This implies that there are $\mathcal{ALC}$ knowledge bases that do not have a model that is both finite *and* forest-shaped, and explains why we must choose which of these two properties is guaranteed by our construction.

We can modify the algorithm from Definition 4.2 to additionally deal with a normalised $\mathcal{ALC}$ TBox $\mathcal{T}$ simply by substituting the rules from Figure 4.6 for those from Figure 4.2. For the sake of completeness, we will recapitulate the (modified) definition here.

[4] Recall that, with the addition of ABoxes, we introduced *forest* models as a generalisation of *tree* models (see Section 4.2.1).

Algorithm consistent()
Input: a normalised $\mathcal{ALC}$ KB ($\mathcal{T}$,$\mathcal{A}$)
if expand$(\mathcal{T}, \mathcal{A}) \neq \emptyset$ **then**
  return "consistent"
else
  return "inconsistent"

Algorithm expand()
Input: a normalised $\mathcal{ALC}$ KB ($\mathcal{T}$,$\mathcal{A}$)
if $\mathcal{A}$ is not complete **then**
  select a rule R that is applicable to $\mathcal{A}$ and an assertion
  or pair of assertions α in $\mathcal{A}$ to which R is applicable
  if there is $\mathcal{A}' \in \mathsf{exp}(\mathcal{A}, R, \alpha)$ with expand$(\mathcal{T}, \mathcal{A}') \neq \emptyset$ **then**
    return expand$(\mathcal{T}, \mathcal{A}')$
  else
    return $\emptyset$
else
  if $\mathcal{A}$ contains a clash **then**
    return $\emptyset$
  else
    return $\mathcal{A}$

Fig. 4.7. The tableau algorithm, consistent, for $\mathcal{ALC}$ knowledge base consistency, and the ABox expansion algorithm expand.

Definition 4.9 (Algorithm for $\mathcal{ALC}$ KB consistency)**.** The algorithm consistent for $\mathcal{ALC}$ KB consistency takes as input a normalised $\mathcal{ALC}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ and uses the algorithm expand to apply the rules from Figure 4.6 to $\mathcal{A}$ with respect to the axioms in the TBox $\mathcal{T}$; both algorithms are given in Figure 4.7.

We will now prove that, for any $\mathcal{ALC}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, the algorithm is terminating, sound and complete.

Lemma 4.10 (Termination)**.** *For each $\mathcal{ALC}$ knowledge base $\mathcal{K}$,* consistent$(\mathcal{K})$ *terminates.*

Proof. The proof is very similar to the proof of Lemma 4.4, the only difference being with respect to the third part of the proof that concerns the depth bound on trees in the forest-shaped ABox. Let $m = |\mathsf{sub}(\mathcal{K})|$. By definition of the rules in Figure 4.6, we have $\mathsf{con}_{\mathcal{A}}(a) \subseteq \mathsf{sub}(\mathcal{K})$, and thus there are at most 2^m different such sets.

(i) There can be at most m rule applications in respect of any given individual name (see Lemma 4.4).

(ii) The outdegree of each tree in the forest-shaped ABox is thus bounded by m (see Lemma 4.4).

(iii) Since $\mathsf{con}_{\mathcal{A}}(a) \subseteq \mathsf{sub}(\mathcal{K})$ and $|\mathsf{sub}(\mathcal{K})| = m$, any path along tree individuals in the ABox generated can contain at most 2^m individual names before it contains two different individual names a and b such that b is a descendant of a, $\mathsf{con}_{\mathcal{A}}(a) \supseteq \mathsf{con}_{\mathcal{A}}(b)$, and application of the $\exists$-rule to b and all of its descendants is thus blocked. The depth of each tree in the forest-shaped ABox is thus bounded by 2^m.

The second and third properties imply that only finitely many new individual names can be generated, and thus the first property yields termination. □

Lemma 4.11 (Soundness)**.** *If* $\mathsf{consistent}(\mathcal{K})$ *returns "consistent", then* $\mathcal{K}$ *is consistent.*

Proof. As in the proof of Lemma 4.5, we use the complete and clash-free ABox $\mathcal{A}'$ returned by $\mathsf{expand}(\mathcal{K})$ to construct a suitable model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{K}$, with the only additional difficulty being how to deal with blocked individual names. This can most easily be achieved in two steps: first by constructing a new ABox $\mathcal{A}''$ that contains those axioms in $\mathcal{A}'$ that do not involve blocked individual names, plus a new "loop-back" role assertion $(a, b') : r$ to replace each $(a, b) : r \in \mathcal{A}'$ in which a is not blocked and b is blocked by b'; and second by using $\mathcal{A}''$ to construct a model of $\mathcal{K}$.

We construct $\mathcal{A}''$ as follows:

$$\begin{aligned}\mathcal{A}'' = {} & \{a : C \mid a : C \in \mathcal{A}' \text{ and } a \text{ is not blocked}\} \cup \\ & \{(a, b) : r \mid (a, b) : r \in \mathcal{A}' \text{ and } b \text{ is not blocked}\} \cup \\ & \{(a, b') : r \mid (a, b) : r \in \mathcal{A}',\ a \text{ is not blocked and } b \text{ is blocked by } b'\}.\end{aligned}$$

It is not hard to see that $\mathcal{A} \subseteq \mathcal{A}''$ because $\mathcal{A} \subseteq \mathcal{A}'$ and, for all assertions $a : C$ and $(a, b) : r$ in $\mathcal{A}$, both a and b are root individuals and so can never be blocked. Note that indeed none of the individual names occurring in $\mathcal{A}''$ is blocked. For concept assertions $a : C$ this is the case by definition. For role assertions, we need to consider two cases. In the first case, if $(a, b) : r \in \mathcal{A}'$ and b is not blocked, then obviously a also cannot be blocked since successors of blocked individual names are also blocked. In the second case, it is sufficient to show that b' is not blocked. In fact,

if b is blocked, the fact that its predecessor a is not blocked implies that b is blocked by some predecessor b' of b in $\mathcal{A}'$. Since either $b' = a$ or b' is a predecessor of a, the fact that a is not blocked implies that b' cannot be blocked.

The following Property (P2) is an immediate consequence of the definition of $\mathcal{A}''$ and will be used repeatedly:

$$\mathsf{con}_{\mathcal{A}''}(a) = \mathsf{con}_{\mathcal{A}'}(a). \tag{P2}$$

Since $\mathcal{A}'$ is clash-free, $\mathcal{A}''$ is also clash-free: Property (P2) implies that if $\mathcal{A}''$ contains a clash, then so does $\mathcal{A}'$. Moreover, $\mathcal{A}'$ being complete implies that $\mathcal{A}''$ is also complete:

- For the $\sqcap$-rule, if $a : C \sqcap D \in \mathcal{A}''$, then Property (P2) implies $a : C \sqcap D \in \mathcal{A}'$. Completeness of $\mathcal{A}'$ implies that $\{a : C, a : D\} \subseteq \mathcal{A}'$ and then Property (P2) implies $\{a : C, a : D\} \subseteq \mathcal{A}''$.
- Analogous arguments hold for the $\sqcup$- and $\sqsubseteq$-rules and are left to the reader as a useful exercise.
- For the $\exists$-rule, if $a : \exists r.C \in \mathcal{A}''$, then $a : \exists r.C \in \mathcal{A}'$ and a is not blocked in $\mathcal{A}'$; hence there is a b such that $\{(a, b) : r, b : C\} \subseteq \mathcal{A}'$ (otherwise $\mathcal{A}'$ would not be complete). We distinguish two cases:
  - If b is not blocked, then $\{(a, b) : r, b : C\} \subseteq \mathcal{A}''$.
  - If b is blocked, the fact that its predecessor a is not blocked implies that b is blocked by some b' in $\mathcal{A}'$, and that b' is not blocked (see the argument given for this fact above). Hence $(a, b') : r \in \mathcal{A}''$, and $\mathsf{con}_{\mathcal{A}'}(b) \subseteq \mathsf{con}_{\mathcal{A}'}(b')$ which, together with Property (P2), yields $b' : C \in \mathcal{A}''$. We therefore have $\{(a, b') : r, b' : C\} \subseteq \mathcal{A}''$.

  In both cases, the $\exists$-rule is not applicable in $\mathcal{A}''$.
- For the $\forall$-rule, if $\{a : \forall r.C, (a, b') : r\} \subseteq \mathcal{A}''$, then $a : \forall r.C \in \mathcal{A}'$ and neither a nor b' is blocked in $\mathcal{A}'$. We distinguish two cases:
  - If $(a, b') : r \in \mathcal{A}'$, then $b' : C \in \mathcal{A}'$ (since $\mathcal{A}'$ is complete), and Property (P2) implies $b' : C \in \mathcal{A}''$.
  - If $(a, b') : r \notin \mathcal{A}'$, then there is a b such that $(a, b) : r \in \mathcal{A}'$, with b blocked by b' in $\mathcal{A}'$, and $b : C \in \mathcal{A}'$ (since $\mathcal{A}'$ is complete). Then the definition of blocking implies $b' : C \in \mathcal{A}'$, and Property (P2) yields $b' : C \in \mathcal{A}''$.

  In both cases, the $\forall$-rule is not applicable in $\mathcal{A}''$.

We can construct an interpretation $\mathcal{I}$ from $\mathcal{A}''$ exactly as in the proof of

Lemma 4.5:

$$\begin{aligned}
\Delta^{\mathcal{I}} &= \{a \mid a \text{ is an individual name occurring in } \mathcal{A}''\},\\
a^{\mathcal{I}} &= a \quad \text{for each individual name } a \text{ occurring in } \mathcal{A}'',\\
A^{\mathcal{I}} &= \{a \mid A \in \mathsf{con}_{\mathcal{A}''}(a)\} \quad \text{for each concept name } A \text{ occurring in } \mathcal{A}'',\\
r^{\mathcal{I}} &= \{(a,b) \mid (a,b) : r \in \mathcal{A}''\} \quad \text{for each role } r \text{ occurring in } \mathcal{A}''.
\end{aligned}$$

From Definition 2.7, $\mathcal{I}$ is a model of $\mathcal{K}$ if it is a model of both $\mathcal{T}$ and $\mathcal{A}$. The proof that $\mathcal{I}$ is a model of $\mathcal{A}''$ and hence of $\mathcal{A}$ is exactly as for Lemma 4.5. In particular, we can show that $a : C \in \mathcal{A}''$ implies $a^{\mathcal{I}} \in C^{\mathcal{I}}$ via an induction on the structure of concepts, as a consequence of $\mathcal{A}''$ being complete and clash-free. From Definition 2.4, it is a model of $\mathcal{T}$ if it satisfies each GCI in $\mathcal{T}$. For each GCI $\top \sqsubseteq D \in \mathcal{T}$,[5] and each individual name a occurring in $\mathcal{A}''$, we have $a : D \in \mathcal{A}''$ (otherwise the $\sqsubseteq$-rule would be applicable) and, as $\mathcal{I}$ is a model of $\mathcal{A}''$, we have $a = a^{\mathcal{I}} \in D^{\mathcal{I}}$. Since a was an arbitrary element of $\Delta^{\mathcal{I}}$, this shows that $\Delta^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ as required. □

Lemma 4.12 (Completeness)**.** *If $\mathcal{K}$ is consistent, then* $\mathsf{consistent}(\mathcal{K})$ *returns "consistent".*

Proof. As in the proof of Lemma 4.6, recursive applications of expand preserve consistency. Blocking makes no difference – it only means that the construction will eventually terminate (as per Lemma 4.10) – so the only difference is with respect to the addition of the $\sqsubseteq$-rule, and this is rather trivial: if $a : C \in \mathcal{A}$ and $\top \sqsubseteq D \in \mathcal{T}$, then Definition 2.4 implies that $a^{\mathcal{I}} \in D^{\mathcal{I}}$ in any model $\mathcal{I}$ of $(\mathcal{T}, \mathcal{A})$, so $\mathcal{I}$ is still a model of $(\mathcal{T}, \mathcal{A} \cup \{a : D\})$. □

Theorem 4.13. *The algorithm presented in Definition 4.9 is a decision procedure for the consistency of $\mathcal{ALC}$ knowledge bases.*

Proof. This follows directly from Lemmas 4.10, 4.11 and 4.12. □

Next, let us discuss the complexity of this algorithm. First, note that the transformation of GCIs into the form $\top \sqsubseteq D$ at most doubles the size of the input knowledge base. Next, we have explained in the proof of Lemma 4.10 that the algorithm generates new individual names that form trees in the forest-shaped ABox whose outdegree is bounded by m, and whose depth is bounded by 2^m, for m the number of subconcepts in the input $\mathcal{K}$. As a consequence, the algorithm presented

[5] Remember that, at the beginning of this section, we assume that all our GCIs are of this form, which is without loss of generality thanks to Lemma 2.16.

requires space that is double exponential in the size of the input knowledge base. This is clearly suboptimal since deciding $\mathcal{ALC}$ consistency with respect to general TBoxes is known to be an ExpTime-complete problem (see Section 5.4 for pointers to the literature), and indeed optimal decision procedures exist, even tableau-based ones, the first one presented in [DGM00]. In [GN13], a more economical form of blocking is used to ensure termination, and so-called global cashing is used to deal with nondeterminism, resulting in a conceptually relatively simple ExpTime tableau algorithm for $\mathcal{ALC}$ with general knowledge bases.

4.3 A tableau algorithm for $\mathcal{ALCIN}$

The algorithm described in Section 4.2.3 can be extended to deal with a wide range of additional constructors; this typically involves modifying existing or adding new expansion rules, and may also require the modification of other parts of the algorithm, such as the definitions of clash-free and blocking. In this section we will consider the changes necessary to deal with $\mathcal{ALCIN}$, which extends $\mathcal{ALC}$ with inverse roles and number restrictions. This is an interesting case for several reasons: inverse roles mean that tree individuals can influence their predecessors as well as their successors; number restrictions mean that we need to deal with equality (of individual names); and the combination of the two means that the logic no longer has the finite model property (see Theorem 3.19), which means that blocks must be assumed to represent infinitely repeating rather than cyclical models.

In the following sections we will discuss these issues in more detail, but still not on a completely formal level. Instead, we will only sketch the ideas behind correctness proofs. The interested reader is referred to [HS04] for details and full proofs.

4.3.1 Inverse roles

In $\mathcal{ALCI}$, roles are no longer restricted to being role names, but can also be inverse roles as per Definition 2.19. Consequently, tree individuals can influence not only their successors (as in the case of $\mathcal{ALC}$), but also their predecessors. For example, if $\{a : \forall r.C, b : \forall r^-.D, (a, b) : r\} \subseteq \mathcal{A}$, then we can infer not only that $\mathcal{A} \models b : C$, due to the interaction between $a : \forall r.C$ and $(a, b) : r$, but also that $\mathcal{A} \models a : D$, due to the interaction between $b : \forall r^-.D$ and the (only implicitly present) role assertion $(b, a) : r^-$.

To make it easier to deal with inverse roles, we define a function Inv

$\exists$-rule: if 1. $a : \exists r.C \in \mathcal{A}$,
2. there is no b such that b is an r-neighbour of a with $b : C \in \mathcal{A}$, and
3. a is not blocked,
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{(a, d) : r, d : C\}$, where d is new in $\mathcal{A}$

$\forall$-rule: if 1. $a : \forall r.C \in \mathcal{A}$, b is an r-neighbour of a, and
2. $b : C \notin \mathcal{A}$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \{b : C\}$

Fig. 4.8. $\exists$- and $\forall$-rules for $\mathcal{ALCI}$.

that allows us to "flip" backwards and forwards between a role and its inverse, so we can avoid the need to consider semantically equivalent roles such as r and $(r^-)^-$, and we introduce the notion of an r-neighbour, so we can avoid the need to consider semantically equivalent role assertions such as $(a, b) : r$ and $(b, a) : r^-$. We define Inv as follows:

$$\mathsf{Inv}(r) = \begin{cases} r^- & \text{if } r \in \mathbf{R}, \\ s & \text{if } r = s^- \text{ and } s \in \mathbf{R}; \end{cases}$$

and we say that an individual name b is an *r-neighbour* of an individual name a in an ABox $\mathcal{A}$ if either $(a, b) : r \in \mathcal{A}$ or $(b, a) : \mathsf{Inv}(r) \in \mathcal{A}$. In our example ABox $\mathcal{A}$ above, b is an r-neighbour of a and a is an r^--neighbour of b.

We can then modify the definitions of the $\exists$- and $\forall$-rules so as to allow for inverse roles simply by referring to "an r-neighbour b of a" (see Fig. 4.8). Note that, as in $\mathcal{ALC}$, successor and predecessor relationships depend on the structure of the tree-shaped parts of the ABox, whereas a neighbour can be a successor, a predecessor or neither (i.e., when two root individuals are neighbours).

In addition, to ensure soundness, we need to modify the definition of blocking (see Definition 4.8).

Definition 4.14 (Equality blocking)**.** An individual name b in an $\mathcal{ALCI}$ ABox $\mathcal{A}$ is *blocked by an individual name a* if

- a is an ancestor of b, and
- $\mathsf{con}_{\mathcal{A}}(a) = \mathsf{con}_{\mathcal{A}}(b)$.

An individual name b is *blocked in $\mathcal{A}$* if it is blocked by some individual name a, or if one or more of its ancestors is blocked in $\mathcal{A}$. When it is clear from the context, we may not mention the ABox explicitly; e.g., we may simply say that b is *blocked*.

For $\mathcal{ALC}$, it sufficed that $\mathsf{con}_{\mathcal{A}}(b) \subseteq \mathsf{con}_{\mathcal{A}}(a)$ for b to be blocked by a; in the presence of inverse roles, we require that $\mathsf{con}_{\mathcal{A}}(b) = \mathsf{con}_{\mathcal{A}}(a)$; i.e., with inverse roles we use *equality blocking* rather than *subset blocking*. This is because, if b is blocked by a, b is an r-successor of c and c is not blocked, then we replace $(c, b) : r$ with $(c, a) : r$ when constructing the ABox $\mathcal{A}''$ in the proof of Lemma 4.11. This is harmless in $\mathcal{ALC}$, but in $\mathcal{ALCI}$ it makes c an $\mathsf{Inv}(r)$-neighbour of a, and the $\forall$-rule might be applicable to an assertion of the form $a : \forall\, \mathsf{Inv}(r).C$ if $\forall\, \mathsf{Inv}(r).C \in \mathsf{con}_{\mathcal{A}'}(a) \setminus \mathsf{con}_{\mathcal{A}'}(b)$; $\mathcal{A}''$ might thus be incomplete, and the algorithm could return "consistent" when $\mathcal{A}$ is inconsistent (i.e., the algorithm would be unsound).

For example, consider the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where

$$\begin{aligned} \mathcal{T} &= \{\top \sqsubseteq \exists r.C, \top \sqsubseteq \forall r^-.(\forall r^-.\neg C)\}, \\ \mathcal{A} &= \{a : C\}. \end{aligned}$$

With subset blocking, the modified expansion rules would construct a complete and clash-free ABox $\mathcal{A}'$ with $(a, x) : r \in \mathcal{A}'$, $\mathsf{con}_{\mathcal{A}'}(x) = \{C, \exists r.C, \forall r^-.(\forall r^-.\neg C)\}$, $\mathsf{con}_{\mathcal{A}'}(a) = \mathsf{con}_{\mathcal{A}'}(x) \cup \{\forall r^-.\neg C\}$ and x blocked by a. However, the construction of $\mathcal{A}''$ would replace $(a, x) : r$ with $(a, a) : r$, and the resulting ABox would no longer be complete as the $\forall$-rule would be applicable to $a : \forall r^-.C$ and $(a, a) : r$. Moreover, applying this rule would add $a : \neg C$, resulting in a clash – in fact it is easy to see that $\mathcal{K}$ is inconsistent.

The use of equality blocking ensures that, in the $\mathcal{ALCI}$ version of the proof of Lemma 4.11, $\mathcal{A}''$ is still complete. This is because, if b is an r-successor of a, and b is blocked by b', then $\forall\, \mathsf{Inv}(r).C \in \mathsf{con}_{\mathcal{A}'}(b')$ implies that $\forall\, \mathsf{Inv}(r).C \in \mathsf{con}_{\mathcal{A}'}(b)$, and the completeness of $\mathcal{A}'$ implies that $a : C \in \mathcal{A}'$. Moreover, we can adapt the model construction part of the proof simply by using the notion of an r-neighbour when constructing role interpretations.

In the above example, $\mathsf{con}_{\mathcal{A}'}(x) \neq \mathsf{con}_{\mathcal{A}'}(a)$, so x is not equality-blocked by a, and the construction continues with the addition of $(x, y) : r$ and, eventually, $y : \forall r^-.(\forall r^-.\neg C)$ to $\mathcal{A}'$. This will in turn lead to the addition of $x : \forall r^-.\neg C$, at which point $\mathsf{con}_{\mathcal{A}'}(x) = \mathsf{con}_{\mathcal{A}'}(a)$, and x is equality-blocked by a. However, the $\forall$-rule is now applicable to $x : \forall r^-.C$ and $(a, x) : r$, and applying this rule adds $a : \neg C$, resulting in a clash.

4.3.2 Number restrictions

With the introduction of number restrictions (see Section 2.5.2) it becomes necessary to deal with (implicit) equalities and inequalities between individual names. For example, if

$$\{(a, x) : r, (a, y) : r, a : (\leqslant 1\, r)\} \subseteq \mathcal{A},$$

then we can infer that x and y are equal, i.e., that in every model $\mathcal{I}$ of $\mathcal{A}$, $x^{\mathcal{I}} = y^{\mathcal{I}}$. Note that in $\mathcal{ALC}$ there is no way to enforce such an equality: for every $\mathcal{ALC}$ knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, and individual names a and b, there exists a model $\mathcal{I}$ of $\mathcal{K}$ in which $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.[6] Similarly, given an assertion of the form $a : (\geqslant n\, r) \in \mathcal{A}$, we can infer that a has at least n r-successors $x_1, \ldots, x_n$ that are pairwise unequal (necessarily interpreted as different elements of the domain).

We can explicate such equalities and inequalities by extending the definition of an ABox to include equality and inequality assertions of the form $a = b$ and $a \neq b$, where a and b are individual names. The semantics of these assertions is straightforward: an interpretation $\mathcal{I}$ satisfies an equality assertion $a = b$ if $a^{\mathcal{I}} = b^{\mathcal{I}}$, and it satisfies an inequality assertion $a \neq b$ if $a^{\mathcal{I}} \neq b^{\mathcal{I}}$. We will use such assertions in our algorithm, but we can assume without loss of generality that they are not present in the ABox of the input knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$: an inequality $a \neq b \in \mathcal{A}$ can be replaced with assertions $a : C$ and $b : \neg C$, where C is new in $\mathcal{K}$, and an equality $a = b \in \mathcal{A}$ can be eliminated by rewriting $\mathcal{A}$ so as to replace all occurrences of b with a (or vice versa) – for example, if b is replaced with a, then $b : C$ would be rewritten as $a : C$ and $(b, d) : r$ would be rewritten as $(a, d) : r$. We will use $\mathcal{A}[b \mapsto a]$ to denote the ABox obtained by replacing each occurrence of b in $\mathcal{A}$ with a.

As usual, the $\mathcal{ALCN}$ expansion rules will only deal with non-negated concepts, and we therefore need to extend our transformation into NNF to deal with negated number restrictions as follows:

$$\begin{array}{lcl} \neg(\geqslant n\, r) & \equiv & \begin{cases} \bot & \text{if } n = 0, \\ (\leqslant (n-1)\, r) & \text{otherwise,} \end{cases} \\ \neg(\leqslant n\, r) & \equiv & (\geqslant (n+1)\, r), \end{array}$$

where n is a non-negative number and r is a role.

The idea for a $\geqslant$-rule is quite intuitive: it is applicable to $a : (\geqslant n\, r) \in \mathcal{A}$ if a has *fewer* than n r-successors, and, when applied, the rule adds n new r-successors of a. Similarly, it is not hard to see that we need

[6] See also Proposition 3.3 in Chapter 3, where it is shown that $\mathcal{ALCN}$ is strictly more expressive than $\mathcal{ALC}$.

a $\leqslant$-rule that is applicable to $a : (\leqslant n\, r) \in \mathcal{A}$ if a has *more* than n r-successors, and when applied, it *merges* two of a's r-successors using the rewriting procedure described above, i.e., when merging b_1, b_2, being two of a's r-successors, a $\leqslant$-rule would return $\mathcal{A}[b_2 \mapsto b_1]$. Of course, this $\leqslant$-rule is nondeterministic: it nondeterministically selects two of a's r-successors and merges them. Moreover, in contrast to the other rules we have discussed, the $\leqslant$-rule does not strictly expand the ABox: it merges one individual name into another, which changes and/or removes ABox assertions. For example, if $\mathcal{A} = \{(a, x) : r, (a, y) : r, x : C, y : D, a : (\leqslant 1\, r)\}$, then applying the $\leqslant$-rule to $a : (\leqslant 1\, r)$ and merging y into x will result in the ABox $\mathcal{A}' = \{(a, x) : r, x : C, x : D, a : (\leqslant 1\, r)\}$.

Ensuring termination becomes much more problematical when we no longer have a monotonically growing ABox (see proof of Lemma 4.10): even if we can establish an upper bound on the size of the ABox, non-termination could result from repeated expansion and contraction of the ABox. Indeed, it is easy to see that conflicting number restrictions could lead to such non-termination; e.g., if $\mathcal{A} = \{a : (\geqslant 2\, r), a : (\leqslant 1\, r)\}$, then the $\geqslant$- and $\leqslant$-rules could be used to repeatedly add and merge r-successors of a. Moreover, a more insidious form of the problem can arise when tree individuals are merged with root individuals; consider, for example, the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where

$$\begin{aligned} \mathcal{T} &= \{\top \sqsubseteq \exists r.A\}, \\ \mathcal{A} &= \{(a, a) : r, a : (\leqslant 1\, r)\}. \end{aligned}$$

We can use $\sqsubseteq$- and $\exists$-rule applications to construct an ABox:

$$\mathcal{A}_1 = \mathcal{T}(\{(a, a) : r, a : (\leqslant 1\, r), (a, x) : r, x : A, (x, y) : r, y : A\}),$$

where $\mathcal{T}(\mathcal{A})$ is shorthand for the ABox resulting from exhaustive application of the $\sqsubseteq$-rule to $(\mathcal{T}, \mathcal{A})$.[7] The $\leqslant$-rule can now be used to merge x into a to give

$$\mathcal{A}_2 = \mathcal{T}(\{(a, a) : r, a : (\leqslant 1\, r), a : A, (a, y) : r, y : A\}),$$

after which applications of the $\exists$- and $\sqsubseteq$-rules lead to the ABox

$$\mathcal{A}_3 = \mathcal{T}(\{(a, a) : r, a : (\leqslant 1\, r), a : A, (a, y) : r, y : A, (y, z) : r, z : A\}).$$

We can now merge y into a, producing an ABox that is isomorphic to $\mathcal{A}_2$ (i.e., they are identical but for individual names), and the process can be repeated indefinitely.[8] Please note that the expansion rules,

[7] We will sometimes use this notation to make examples more readable.

[8] This kind of example has, for obvious reasons, sometimes been called a "yo-yo".

$\geqslant$-rule: if 1. $a : (\geqslant n\, r) \in \mathcal{A}$, a is not blocked, and
2. there do not exist distinct $b_1, \ldots, b_n$ such that $(a, b_i) : r \in \mathcal{A}$ for $1 \leq i \leq n$
then $\mathcal{A} \longrightarrow \mathcal{A} \cup \bigcup_{1 \leq i \leq n} \{(a, d_i) : r\} \cup \bigcup_{1 \leq i < j \leq n} \{d_i \neq d_j\}$
where $d_1, \ldots, d_n$ are new in $\mathcal{A}$.

$\leqslant$-rule: if 1. $a : (\leqslant n\, r) \in \mathcal{A}$, and
2. there exist distinct $b_0 \ldots b_n$ such that $(a, b_i) : r \in \mathcal{A}$ for $0 \leq i \leq n$
then $\mathcal{A} \longrightarrow (\mathsf{prune}(\mathcal{A}, b_j))[b_j \mapsto b_i] \cup \{b_i = b_j\}$
for some $0 \leq i < j \leq n$,
such that, if b_j is a root individual, then so is b_i.

Fig. 4.9. The $\geqslant$- and $\leqslant$-rules for $\mathcal{ALCN}$.

when applied in a different order to this example, can result in a clash-free and complete ABox but, so far, we have striven to design tableau algorithms that are sound, complete and terminating *regardless* of the order in which rules are applied, i.e., without imposing any priorities on rule application.

In order to regain termination, both the $\geqslant$- and $\leqslant$-rules are augmented as shown in Figure 4.9. The $\geqslant$-rule is augmented so that, as well as adding new r-successors, it adds pairwise inequality assertions between the newly added individual names, the purpose of which is to prevent the merging of individual names added by the same $\geqslant$-rule application. The introduction of inequality assertions also necessitates the definition of a new kind of clash: in addition to the condition from Definition 4.1, an ABox $\mathcal{A}$ contains a clash if, for some individual name a, $a \neq a \in \mathcal{A}$; this would require that $a^{\mathcal{I}} \neq a^{\mathcal{I}}$, which clearly precludes any satisfying interpretation. This new clash condition means that, although it is still possible to apply the $\leqslant$-rule in an "obviously silly" way, i.e., by merging two individual names b_1 and b_2 with $b_1 \neq b_2 \in \mathcal{A}$, such a rule application will immediately result in a clash of the form $b_1 \neq b_1$ in $\mathcal{A}[b_2 \mapsto b_1]$. This ensures that the $\geqslant$-rule can be applied at most once in respect of any given ABox assertion.

The extended $\geqslant$-rule and clash conditions allow us to bound the number of successors that can be added to any given individual name by applications of the $\exists$- and $\geqslant$-rules. However, unlike in the case of $\mathcal{ALC}$, successors can also be added by the $\leqslant$-rule: in the above yo-yo example, y is originally added as a successor of x, but subsequently becomes a successor of a when x is merged into a. To address this issue, the $\leqslant$-rule is augmented so as to use a procedure called *pruning* to remove

all descendants of an individual name before it is merged into another individual name, and to add an equality assertion that allows us to "remember" which individual names have been merged.[9] More precisely, $\mathsf{prune}(\mathcal{A}, a)$ is defined to be the ABox that results from removing all assertions of the form $x : C$ or $(y, x) : r$ from $\mathcal{A}$, where x is a descendant of a. The $\leqslant$-rule also ensures that a root individual is never merged into a tree individual: this could cause the ABox to lose its forest shape, which is a fundamental assumption underlying the algorithm, and could even result in the entire ABox being removed by pruning.

We can now adapt the termination argument from Lemma 4.10 to establish a bound on the number of individual names that can be added to $\mathcal{A}$ – which clearly also bounds the size of $\mathcal{A}$. There can still be at most $m = |\mathsf{sub}(\mathcal{K})|$ applications of the $\exists$- and $\geqslant$-rules in respect of any given individual name. For the $\exists$-rule, if assertions $(a, x) : r$ and $x : C$ are added to $\mathcal{A}$ as a result of the $\exists$-rule being applied to an assertion $a : \exists r.C \in \mathcal{A}$, and x is subsequently merged into y, then it must be the case that $(a, y) : r \in \mathcal{A}$; otherwise the $\leqslant$-rule would not have been applicable; thus, after the merge, $\{(a, y) : r, y : C\} \subseteq \mathcal{A}$, and the $\exists$-rule cannot be applied again to $a : \exists r.C$. For the $\geqslant$-rule, the inequality assertions mean that merging two individual names added by any given rule application will immediately result in a clash, and hence the rule cannot be applied twice in respect of the same assertion. Thus $\exists$- and $\geqslant$-rules can add at most $m \times n$ successors to an individual name a, where n is the largest number occurring in a $\geqslant$-restriction in $\mathcal{A}$. Moreover, because of pruning, these are the *only* individual names that can ever be successors of a. Thus the number of individual names that can be added at depth d of each tree in the forest-shaped ABox is bounded by mn^d (assuming $d = 0$ for root individuals), and when combined with the 2^m depth bound due to blocking this gives a bound of mn^{2^m} on the number of individual names that can be added to any such tree.

For soundness, the proof is similar to the one for Lemma 4.11, but the construction of $\mathcal{A}''$ must be modified so that it leads to a complete and clash-free ABox. For example, if $\{a : (\geqslant 2\, r), (a, x) : r, (a, y) : r, x \neq y\} \subseteq \mathcal{A}'$, and both x and y are blocked by z, then replacing $(a, x) : r$ and $(a, y) : r$ with $(a, z) : r$ in $\mathcal{A}''$ would effectively merge x and y into z, resulting in a clash (or, equivalently, in the applicability of the $\geqslant$-rule to $a : (\geqslant 2\, r)$). However, we can extend $\mathcal{A}'$ by adding copies of blocking

[9] The equality assertion is not used during expansion, but it will be useful in the completeness proof to construct a model of the input knowledge base from a complete and clash-free ABox.

individual names for each of the individual names that they block; for example, if x is blocked by z, then we can introduce a new individual name xz and add concept assertions $xz : C$ for each concept C with $z : C \in \mathcal{A}'$ and role assertions $(xz, y) : r$ for each role r and individual name y with $(z, y) : r \in \mathcal{A}'$. It is easy to see that $\mathcal{A}''$ is still complete and clash-free (any clash or rule applicable to one of the copy individual names would have applied to the individual name from which it was copied), and we can proceed with the construction of $\mathcal{A}''$ as per Lemma 4.11, except that when x is blocked by z we treat it as being blocked by xz. Finally, we can copy the equality assertions from $\mathcal{A}'$ to $\mathcal{A}''$ and use these in the model construction to ensure that each individual name occurring in $\mathcal{A}$ is appropriately interpreted.

The completeness proof only requires a straightforward extension of the case analysis from Lemma 4.6 to include the $\geqslant$- and $\leqslant$-rules. Remember that, like the $\sqcup$-rule, the $\leqslant$-rule is nondeterministic, and a knowledge base is consistent if and only if at least one such selection yields a consistent knowledge base. As with the $\sqcup$-rule, the algorithm expand from Figure 4.3 will recursively explore all possible ways of applying the $\leqslant$-rule, the number of which can escalate rapidly with larger number restrictions; e.g., if $a : (\leqslant 5\, r) \in \mathcal{A}$ and a has ten r-successors in $\mathcal{A}$, then there are

$$\frac{10!}{(5-1)!(10-(5-1))!} = 210$$

different ways of merging these successors so as to satisfy the number restriction, and this increases to 167,960 for $a : (\leqslant 10\, r)$ with twenty r-successors.

4.3.3 Combining inverse roles and number restrictions

It might seem that we can combine inverse roles with number restrictions simply by modifying the $\geqslant$- and $\leqslant$-rules from Figure 4.9 such that the b_i are r-neighbours of a. However, interactions between inverse roles and number restrictions introduce some additional difficulties that require careful handling.

First, the merging performed by the $\leqslant$-rule could destroy the forest shape of the ABox. Consider, for example, an ABox

$$\mathcal{A} \supseteq \{(w, x) : r, (x, y) : r^-, (y, z) : r, y : (\leqslant 1\, r)\},$$

where w, x, y and z are tree individuals, and x, y and z are successors

of, respectively, w, x and y. Both x and z are r-neighbours of y, so the $\leqslant$-rule is applicable to $y : (\leqslant 1\, r)$; however, merging x into z would result in the ABox

$$\mathcal{A}' \supseteq \{(w, z) : r, (z, y) : r^-, (y, z) : r, y : (\leqslant 1\, r)\},$$

in which the tree individuals are no longer arranged in a tree shape: even if we ignore the semantically redundant assertion $(z, y) : r^-$, the individual name y has no predecessor. In order to deal with this problem, we can modify the $\leqslant$-rule so that it never merges an individual name into one of its descendants and, although not strictly necessary, the rule can also remove semantically redundant role assertions that result from merging an individual name into one of its ancestors, as these would complicate the model construction in the soundness proof. In the above example, this would result in z being merged into x, and the removal of the rewritten role assertion $(y, x) : r$ (which is semantically equivalent to $(x, y) : r^-$), to give

$$\mathcal{A} \supseteq \{(w, x) : r, (x, y) : r^-, y : (\leqslant 1\, r)\}.$$

Second, the construction of a finite model used in the proof of Lemma 4.11 clearly cannot work, as $\mathcal{ALCI\!N}$ does not have the finite model property (see Theorem 3.19). In particular, if $(y, z) : r \in \mathcal{A}'$ and z is blocked by x, then the construction of $\mathcal{A}''$ replaces $(y, z) : r$ with $(y, x) : r$ (in the proof of Lemma 4.11) or $(y, zx) : r$ (in the adapted construction for $\mathcal{ALCN}$); but in either case, if $x : (\leqslant 1\, r^-) \in \mathcal{A}'$ and x (and hence also zx) already has an unblocked r^--neighbour in $\mathcal{A}'$, then it would get a second one. As a consequence, $\mathcal{A}''$ would no longer be complete. Consider, for example, the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where

$$\begin{aligned} \mathcal{T} &= \{\top \sqsubseteq \exists r.A, \top \sqsubseteq (\leqslant 1\, r^-)\}, \\ \mathcal{A} &= \{a : \neg A\}. \end{aligned}$$

We can use expansion rule applications to generate an ABox

$$\mathcal{A}' = \mathcal{T}(\{a : \neg A, (a, x) : r, x : A, (x, y) : r, y : A\}).$$

The ABox $\mathcal{A}'$ is complete, with y being blocked by x, but if we replace $(x, y) : r$ with $(x, x) : r$ in the construction of $\mathcal{A}''$, then $\mathcal{A}''$ is no longer complete as both x and a are r^--neighbours of x, and the $\leqslant$-rule would be applicable to $x : (\leqslant 1\, r^-) \in \mathcal{A}''$. Moreover, applying the rule would merge x into a, resulting in a clash ($\{a : A, a : \neg A\} \subseteq \mathcal{A}''$). The same problem arises if we use a copy yx of the blocking node; we leave it as an exercise for the reader to work through the example.

This problem can be overcome by using a stronger *pairwise* blocking condition[10] which ensures that $\mathcal{A}'$ can be used to construct a (possibly infinite) forest-shaped $\mathcal{A}''$ that is complete and clash-free and from which we can construct a (possibly infinite) model in the usual way.

Definition 4.15 (Pairwise blocking)**.** An individual name b is *blocked by an individual name* a in an $\mathcal{ALCIN}$ ABox $\mathcal{A}$ if, for some role r, b has ancestors a', a and b' such that:

(i) a is an r-successor of a' and b is an r-successor of b';
(ii) $\mathsf{con}_{\mathcal{A}}(a) = \mathsf{con}_{\mathcal{A}}(b)$ and $\mathsf{con}_{\mathcal{A}}(a') = \mathsf{con}_{\mathcal{A}}(b')$.

An individual name b is *blocked in* $\mathcal{A}$ if it is blocked by some individual name a, or if one or more of its ancestors is blocked in $\mathcal{A}$. When it is clear from the context, we may not mention the ABox explicitly; e.g., we may simply say that b is *blocked*.

When $\mathcal{A}'$ contains blocked individual names, we can construct a forest-shaped ABox by replacing them with a copies of the subtrees rooted in the corresponding blocking individual names. A subtree rooted in a blocking individual name necessarily includes the individual names that it blocks, and so the copying process is infinitely recursive, a procedure that is sometimes referred to as "unravelling" (see Definition 3.21). More precisely, the construction of $\mathcal{A}''$ follows the same pattern as in the proof of Lemma 4.11, but in the situation where $(a, b) : r \in \mathcal{A}'$, with a not blocked and b blocked by b', we add $\{(a, b'') : r\} \cup \mathsf{copy}(b'', b')$ to $\mathcal{A}''$, where b'' is new in $\mathcal{A}''$, and $\mathsf{copy}(x, y)$ is defined as the smallest set that includes:

- $\{x : C\}$ for each concept assertion $y : C \in \mathcal{A}'$;
- $\{(x, z') : r\} \cup \mathsf{copy}(z', z)$ for each role assertion $(y, z) : r \in \mathcal{A}'$, where z is not blocked in $\mathcal{A}'$ and z' is new in $\mathcal{A}''$;
- $\{(x, z') : r\} \cup \mathsf{copy}(z', w)$ for each role assertion $(y, z) : r \in \mathcal{A}'$, where z is blocked by w in $\mathcal{A}'$ and z' is new in $\mathcal{A}''$.

In the above example, where

$$\mathcal{A}' \quad = \quad \mathcal{T}(\{a : \neg A, (a, x) : r, x : A, (x, y) : r, y : A\}),$$

and y is blocked by x, unravelling $(x, y) : r$ would add $\{(x, x') : r\} \cup \mathsf{copy}(x', x)$ to $\mathcal{A}''$, which adds concept assertions such that $\mathsf{con}_{\mathcal{A}''}(x') = \mathsf{con}_{\mathcal{A}'}(x)$ and a role assertion $(x', x'') : r$ such that x'' is new in $\mathcal{A}''$, and

[10] Sometimes called double blocking.

x'' is (recursively) a copy of x. The recursion will result in a complete and clash-free set of assertions[11] that includes an infinite sequence of r-successors, each of which is a copy of x. In general, however, simple equality blocking (see Definition 4.14) is not sufficient to guarantee that we can use unravelling to construct such an ABox (and hence to construct a model). Consider, for example, the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, where

$$\begin{aligned} \mathcal{T} &= \{\top \sqsubseteq \exists r.A, \top \sqsubseteq \exists r^-.\neg A, \top \sqsubseteq (\leqslant 1\, r^-)\}, \\ \mathcal{A} &= \{a : \neg A, (a,a) : r\}. \end{aligned}$$

We can use expansion rule applications to generate an ABox

$$\mathcal{A}' = \mathcal{T}(\{a : \neg A, (a,a) : r, (a,x) : r, x : A, (x,y) : r, y : A\}),$$

in which y is equality blocked by x. Note that the $\exists$-rule is not applicable to $x : \exists r^-.\neg A$, because a is an r-neighbour of x with $a : \neg A \in \mathcal{A}'$. However, when we start unravelling, we replace y with a copy x' of x, and the $\exists$-rule *is* applicable to $x' : \exists r^-.\neg A$, because x is the only r-neighbour of x', and $x : \neg A \notin \mathcal{A}'$. Moreover, applying the $\exists$-rule to $x' : \exists r^-.\neg A$ will add $(x', z) : r^-$ and $z : \neg A$, and the $\leqslant$-rule will merge z into x, revealing a clash – indeed it is easy to see that $\mathcal{K}$ is inconsistent, and that pairwise blocking (rather than equality blocking) is indeed required to detect this.

Pairwise blocking ensures that, when a blocked individual name y is replaced with a copy x' of the individual name x that blocks y, the neighbours of x' are indistinguishable from those of x. This is clearly the case for the successors of x', as these are copies of the successors of x, and pairwise blocking ensures that this is also the case for the predecessors of x' and of x (note that pairwise blocking ensures that both blocked and blocking individual names are tree individuals, and so each has exactly one predecessor, and no neighbours other than its predecessor and successors). Thus if any expansion rule were to be applicable to x', then it would have been applicable to x. Moreover, the construction of $\mathcal{A}''$ cannot introduce a clash, as for each newly introduced individual name x', $\mathsf{con}_{\mathcal{A}''}(x') = \mathsf{con}_{\mathcal{A}'}(x)$ for some individual name x in $\mathcal{A}'$. Thus, if $\mathcal{A}'$ is complete and clash-free, then so is $\mathcal{A}''$.

[11] This set is not strictly speaking an ABox since ABoxes are *finite* sets of assertions, but the semantics is the same.

4.4 Some implementation issues

As we have seen, tableau algorithms prove that a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is consistent by constructing a sequence of ABoxes $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_n$ where $\mathcal{A}_0 = \mathcal{A}$ and each $\mathcal{A}_i$ is obtained from $\mathcal{A}_{i-1}$ by an application of one of the expansion rules. Some of these rules may be nondeterministic, e.g., the $\sqcup$-rule: if $a : C \sqcup D \in \mathcal{A}$, then either $a : C$ or $a : D$ (or both) must be satisfied, and the algorithm may have to make a nondeterministic guess as to which one to add to $\mathcal{A}$. If the first such guess leads to a clash, then the algorithm must backtrack and try each of the other possible choices in turn, with $\mathcal{K}$ being inconsistent only if all such choices lead to a clash. This process is sometimes referred to as *or-branching*. Other rules can cause new individual names to be added to the ABox, e.g., the $\exists$-rule: if $a : \exists r.C \in \mathcal{A}$, then the algorithm may have to add an assertion $b : C$ to $\mathcal{A}$, where b is an individual name that did not previously occur in $\mathcal{A}$. This process is sometimes referred to as *and-branching*.

Both kinds of branching can be a cause of scalability problems in practice: or-branching may lead to the exploration of an infeasibly large number of expansion choices, while and-branching may lead to the construction of an infeasibly large ABox. Modern tableau reasoners include numerous optimisations aimed at curbing both kinds of branching.

In practice, DL reasoners are typically used not to perform single KB consistency tests, but to perform large numbers of reasoning tasks with respect to the same KB. A prominent example is classification: the computation of all subsumption relationships between concept names in the input KB (see Section 2.3). Tableau-based reasoners invariably include optimisations whose goal is to minimise the number of KB consistency tests performed during classification.

A comprehensive survey of these and other optimisation techniques is beyond the scope of this chapter (the interested reader is referred to [THPS07] for such a survey, and to [GHM+14] for some more recent work), but we will briefly discuss some of the most important and widely used techniques.

4.4.1 Or-branching

The technique for dealing with arbitrary GCIs described in Section 4.2.3 is simple, but extremely inefficient in practice. In fact, a GCI of the form $C \sqsubseteq D$ is transformed into the GCI $\top \sqsubseteq D \sqcup \neg C$, and thus for each GCI $C \sqsubseteq D$ in $\mathcal{T}$ and each individual name occurring in the ABox, the $\sqsubseteq$-rule causes an assertion of the form $a : D \sqcup \neg C$ to be added. Given a KB

with a TBox containing only 10 GCIs and an ABox containing only 10 individual names, the $\sqsubseteq$-rule would thus add at least 100 such assertions to the ABox, and as many as 2^{100} different sequences of nondeterministic expansion choices may thus need to be explored. Moreover, this will happen even if the KB falls within a fragment of the logic for which deterministic reasoning is possible (see, e.g., Chapter 6).

Lazy unfolding and *absorption* are optimisation techniques that address this problem; they are among the most important and widely used optimisation techniques for tableau algorithms, and without them tableau algorithms for general knowledge base consistency would be hopelessly impractical. As we saw in Section 4.2.2, acyclic TBox axioms can be dealt with deterministically. This technique does not work for general TBox axioms, but a general TBox $\mathcal{T}$ can be divided into two disjoint subsets $\mathcal{T}_a$ and $\mathcal{T}_g$ such that $\mathcal{T} = \mathcal{T}_a \cup \mathcal{T}_g$ and $\mathcal{T}_a$ is acyclic. The lazy expansion rules from Fig. 4.5 can then be used to deal with axioms in $\mathcal{T}_a$, with the $\sqsubseteq$-rule being used only for axioms in $\mathcal{T}_g$.

Although much more efficient, even this approach may be impractical unless $\mathcal{T}_g$ is small. Absorption is a technique that tries to rewrite axioms so as to increase the size of $\mathcal{T}_a$ and reduce the size of $\mathcal{T}_g$. In its most basic form, absorption rewrites axioms of the form $A \sqcap B \sqsubseteq C$ as $A \sqsubseteq C \sqcup \neg B$. This axiom can then be "absorbed" into another axiom $A \sqsubseteq D \in \mathcal{T}_a$ to give $A \sqsubseteq D \sqcap (C \sqcup \neg B)$, with $A \sqcap B \sqsubseteq C$ then being removed from $\mathcal{T}_g$ – provided that this preserves acyclicity of $\mathcal{T}_a$. Although a disjunction is still present in the axiom $A \sqsubseteq D \sqcap (C \sqcup \neg B)$, lazy unfolding ensures that this disjunction is only introduced for those individual names a such that $a : A$ is in the ABox.

Many refinements and extensions of absorption have been described in the literature. In some respects the more recently developed *hypertableau* algorithm used in the HermiT reasoner can be seen as the ultimate refinement of absorption: the algorithm uses a more complex form of expansion rule that allows for the lazy expansion of all (normalised) axioms.

Even if $\mathcal{T}_g$ is empty, disjunctive concepts in $\mathcal{T}_a$ can still lead to the exploration of large numbers of nondeterministic expansion choices, and pathological cases can arise when inherent unsatisfiability is concealed in subdescriptions. For example, expanding the assertion

$$\begin{aligned} a : \; & (\exists R.(A \sqcap B) \sqcup \exists R.(A \sqcap C)) \sqcap \\ & (\forall R.D_1 \sqcup \forall R.E_1) \sqcap \ldots \sqcap (\forall R.D_n \sqcup \forall R.E_n) \sqcap \\ & (\forall R.(\neg A \sqcap X) \sqcup \forall R.(\neg A \sqcap Y)) \end{aligned}$$

could lead to the fruitless exploration of 2^n possible expansions of $(\forall R.D_1 \sqcup \forall R.E_1) \sqcap \cdots \sqcap (\forall R.D_n \sqcup \forall R.E_n)$ before the inherent unsatisfiability of the first and last conjuncts is discovered. This problem is often addressed by adapting a form of dependency-directed backtracking called *backjumping*.

Backjumping works by labelling concepts with a dependency set indicating the nondeterministic expansion choices on which they depend. When a clash is discovered, the dependency sets of the clashing concepts can be used to identify the most recent nondeterministic expansion where an alternative choice might alleviate the cause of the clash. The algorithm can then "jump back" over intervening nondeterministic expansions *without* exploring any alternative choices.

4.4.2 And-branching

Although blocking ensures that the expansion process terminates, the ABox constructed by the algorithm can in some cases be large enough to cause serious performance problems. This problem is particularly prevalent in cases where the ontology describes structures that are not tree-like and/or where inverse roles are used. For example, in an ontology describing human anatomy, physical connections and part–whole relations between anatomical components are naturally cyclical:

$$\begin{aligned} \mathsf{Head} &\sqsubseteq \exists \mathit{hasPart}.\mathsf{Skull}, \\ \mathsf{Skull} &\sqsubseteq \exists \mathit{hasPart}^{-}.\mathsf{Head}. \end{aligned}$$

The tree-shaped ABox constructed by tableau algorithms can include numerous repetitions of large parts of the intended cyclical model.

One way to address this issue is to optimise blocking conditions so as to halt construction of the ABox at an earlier stage; examples include the use of more fine-grained blocking conditions [HS02] and of speculative blocking conditions that require subsequent checking in order to ensure correctness [GHM10].

Another way to address the same issue is to try to reuse parts of the ABox rather than reconstructing them. For example, if the ABox contains two individual names a and b such that $\mathsf{con}_{\mathcal{A}}(a) = \mathsf{con}_{\mathcal{A}}(b)$, then it might be possible to avoid further expansion of b by reusing the result of expanding a. This kind of technique can be particularly effective if many reasoning tasks are performed with respect to the same KB, for example during classification (see Section 4.4.3), as partial results may be reusable in multiple subsumption tests.

4.4.3 Classification

Classification is a basic reasoning task that is widely used to support ontology engineering, and as a precursor to other reasoning tasks (and optimisations) that exploit the concept hierarchy. Classification could, in the worst case, require $O(n^2)$ subsumption tests, where n is the number of concept names occurring in the TBox, with each subsumption test being transformed into a KB consistency test as described at the beginning of this chapter. However, implementations typically include a range of optimisations that can significantly reduce this number. The most commonly used technique is to construct the hierarchy iteratively, using top-down and bottom-up traversals of the partially constructed hierarchy to determine where to insert each concept name – a technique known as *enhanced traversal* [BFH+94, GHM+12]. Both traversals exploit the transitivity of the subsumption relation in order to avoid performing useless subsumption tests; e.g., if $\mathcal{T} \not\models D \sqsubseteq C$ and $\mathcal{T} \models B \sqsubseteq C$, then we can infer $\mathcal{T} \not\models D \sqsubseteq B$ without performing a test.

Refinements of this basic technique may exploit details of the subsumption reasoning procedure in order to further reduce the number of tests; e.g., when using tableau reasoning, determining $\mathcal{T} \not\models D \sqsubseteq C$ will typically involve the construction of a (partial) model of $D \sqcap \neg C$ that might also be used to prove other non-subsumptions.

Another widely used technique is to exploit more efficient but incomplete or unsound tests in order to avoid invoking a sound and complete tableau-based procedure. A common example is the use of sound but incomplete syntax-based reasoning to identify "obvious" subsumptions; e.g., if $A \sqsubseteq B \sqcap C \in \mathcal{T}$, then we can trivially infer $\mathcal{T} \models A \sqsubseteq B$ and $\mathcal{T} \models A \sqsubseteq C$, and if $\mathcal{T}$ additionally includes $C \sqsubseteq D$, then we can also infer $\mathcal{T} \models A \sqsubseteq D$. This technique is often used in conjunction with enhanced traversal in order to determine a good order in which to insert concept names in the subsumption hierarchy, the goal being to insert a concept name only after all subsuming concept names have already been inserted. Similarly, complete but unsound reasoning techniques can be used to cheaply identify non-subsumptions, an example being the so-called *model merging* technique [BCM+07, Chapter 9].

4.5 Historical context and literature review

Early description logic reasoners such as KL-ONE [BS85], KRYPTON [BFL83], LOOM [Mac91b], CLASSIC [PSMB+91] and BACK [Pel91]

were mainly based on relatively ad-hoc structural subsumption algorithms; see [WS92] for a comprehensive historical account and overview. An alternative approach based on model construction was first introduced by Schmidt-Schauß and Smolka [SS91]; they apparently failed to notice the similarity to the tableau calculus for first-order logic [Smu68], but this was soon pointed out by Donini et al. [DHL+92]. Schmidt-Schauß and Smolka considered only $\mathcal{ALC}$, but the "tableau" technique was soon extended to support a range of constructors including, for example, (qualified) number restrictions [HB91] and concrete domains [BH91]. Moreover, an implementation of one such algorithm in the Kris system showed that, with suitable optimisations, performance on realistic problems could be comparable with or even superior to existing structural approaches [BFH+92].

Initially, most such algorithms and systems, including Kris, considered only concept subsumption or, equivalently, subsumption with respect to an acyclic TBox (see Section 4.2.2). Algorithms for DLs that support general TBoxes and other features that require some form of cycle detection (such as blocking) were soon developed [Baa91, BDS93], but were thought to be impractical due to their high worst-case complexity. However, the FaCT system subsequently demonstrated that a suitably optimised implementation of such a logic could work well in realistic applications [Hor97].

The success of the FaCT system prompted the development of tableau algorithms for increasingly expressive DLs with features such as inverse roles [HS99], qualified number restrictions [HSTT00], complex role inclusion axioms [HS04] and nominals [HS01]. These algorithms were implemented in systems such as FaCT, Racer [HM01], FaCT++ [TH06], Pellet [SPC+07] and HermiT [GHM+14]. The HermiT system is particularly interesting as it uses a so-called hypertableau algorithm in order to reduce the nondeterminism introduced by GCIs [MSH09].

This line of research culminated in the development of $\mathcal{SROIQ}$, a DL that combines all of the above mentioned features [HKS06]. This combination proved to be non-trivial due to complex interactions between inverse roles, number restrictions and nominals, and leads to an increase in complexity from NExpTime to N2ExpTime [Kaz08]. Nevertheless, $\mathcal{SROIQ}$ has been successfully implemented in several of the above mentioned systems, as well as in hybrid systems such as MORe [ACH12] and Konclude [SLG14] that combine tableau with other reasoning techniques, including consequence-based approaches (see Chapter 6); it also forms the basis for the OWL ontology language (see Chapter 8).