9

Implementation and Optimisation Techniques

Ian Horrocks

Abstract

This chapter will discuss the implementation of the reasoning services which form the core of Description Logic based Knowledge Representation Systems. To be useful in realistic applications, such systems need both expressive logics and fast reasoners. As expressive logics inevitably have high worst-case complexities, this can only be achieved by employing highly optimised implementations of suitable reasoning algorithms. Systems based on such implementations have demonstrated that they can perform well with problems that occur in realistic applications, including problems where unoptimised reasoning is hopelessly intractable.

9.1 Introduction

The usefulness of Description Logics (DLs) in applications has been hindered by the basic conflict between expressiveness and tractability. Realistic applications typically require both expressive logics, with inevitably high worst case complexities for their decision procedures, and acceptable performance from the reasoning services. Although the definition of acceptable may vary widely from application to application, early experiments with DLs indicated that, in practice, performance was a serious problem, even for logics with relatively limited expressive powers [Heinsohn *et al.*, 1992].

On the other hand, theoretical work has continued to extend our understanding of the boundaries of decidability in DLs, and has led to the development of sound and complete reasoning algorithms for much more expressive logics. The expressive power of these logics goes a long way towards addressing the criticisms levelled at DLs in traditional applications such as ontological engineering [Doyle and Patil, 1991] and is sufficient to suggest that they could be useful in several exciting new application domains, for example reasoning about DataBase schemata and queries [Calvanese *et al.*, 1998f; 1998a] and providing reasoning support for the so-called

Semantic Web [Decker *et al.*, 2000; Bechhofer *et al.*, 2001b]. However, the worst case complexity of their decision procedures is invariably (at least) exponential with respect to problem size.

This high worst case complexity initially led to the conjecture that expressive DLs might be of limited practical applicability [Buchheit *et al.*, 1993c]. However, although the theoretical complexity results are discouraging, empirical analyses of real applications have shown that the kinds of construct which lead to worst case intractability rarely occur in practice [Nebel, 1990b; Heinsohn *et al.*, 1994; Speel *et al.*, 1995], and experiments with the KRIS system showed that applying some simple optimisation techniques could lead to a significant improvement in the empirical performance of a DL system [Baader *et al.*, 1992a]. More recently the FaCT, DLP and RACER systems have demonstrated that, even with very expressive logics, highly optimised implementations can provide acceptable performance in realistic applications [Horrocks and Patel-Schneider, 1999; Haarslev and Möller, 2001c].[1]

In this chapter we will study the implementation of DL systems, examining in detail the wide range of optimisation techniques that can be used to improve performance. Some of the techniques that will be discussed are completely independent of the logical language supported by the DL and the kind of algorithm used for reasoning; many others would be applicable to a wide range of languages and implementation styles, particularly those using search based algorithms. However, the detailed descriptions of implementation and optimisation techniques will assume, for the most part, reasoning in an expressive DL based on a sound and complete tableaux algorithm.

9.1.1 Performance analysis

Before designing and implementing a DL based Knowledge Representation System, the implementor should be clear about the goals that they are trying to meet and against which the performance of the system will ultimately be measured. In this chapter it will be assumed that the primary goal is utility in realistic applications, and that this will normally be assessed by empirical analysis.

Unfortunately, as DL systems with very expressive logics have only recently become available [Horrocks, 1998a; Patel-Schneider, 1998; Haarslev and Möller, 2001e], there are very few applications that can be used as a source for test data.[2] One application that has been able to provide such data is the European GALEN project, part of which has involved the construction of a large DL Knowledge Base describing

[1] It should be pointed out that experience in this area is still relatively limited.
[2] This situation is changing rapidly, however, with the increasing use of DLs in DataBase and ontology applications.

medical terminology [Rector *et al.*, 1993]. Reasoning performance with respect to this knowledge base has been used for comparing DL systems [Horrocks and Patel-Schneider, 1998b], and we will often refer to it when assessing the effectiveness of optimisation techniques.

As few other suitable knowledge bases are available, the testing of DL systems has often been supplemented with the use of randomly generated or hand crafted test data [Giunchiglia and Sebastiani, 1996b; Heuerding and Schwendimann, 1996; Horrocks and Patel-Schneider, 1998b; Massacci, 1999; Donini and Massacci, 2000]. In many cases the data was originally developed for testing propositional modal logics, and has been adapted for use with DLs by taking advantage of the well know correspondence between the two formalisms [Schild, 1991]. Tests using this kind of data, in particular the test suites from the Tableaux'98 comparison of modal logic theorem provers [Balsiger and Heuerding, 1998] and the DL'98 comparison of DL systems [Horrocks and Patel-Schneider, 1998b], will also be referred to in assessments of optimisation techniques.

9.2 Preliminaries

This section will introduce the syntax and semantics of DLs (full details of which can be found in Chapter 2) and discuss the reasoning services which would form the core of a Description Logic based Knowledge Representation System. It will also discuss how, through the use of *unfolding* and *internalisation*, these reasoning services can often be reduced to the problem of determining the satisfiability of a single concept.

9.2.1 Syntax and semantics

DLs are formalisms that support the logical description of concepts and roles. Arbitrary concept and role descriptions (from now on referred to simply as concepts and roles) are constructed from atomic concept and role names using a variety of concept and role forming operators, the range of which is dependent on the particular logic. In the following discussion we will use C and D to denote arbitrary concepts, R and S to denote arbitrary roles, A and P to denote atomic concept and role names, and n to denote a nonnegative integer.

For concepts, the available operators usually include some or all of the standard logical connectives, *conjunction* (denoted $\sqcap$), *disjunction* (denoted $\sqcup$) and *negation* (denoted $\neg$). In addition, the universal concept *top* (denoted $\top$, and equivalent to $A \sqcup \neg A$) and the incoherent concept *bottom* (denoted $\bot$, and equivalent to $A \sqcap \neg A$) are often predefined. Other commonly supported operators include restricted forms of quantification called *existential role restrictions* (denoted $\exists R.C$) and *universal role*

restrictions (denoted $\forall R.C$). Some DLs also support *qualified number restrictions* (denoted $\leqslant n\,.\mathsf{P}C$ and $\geqslant n\,.\mathsf{P}C$), operators that place cardinality restrictions on the roles relating instances of a concept to instances of some other concept. Cardinality restrictions are often limited to the forms $\leqslant n\,.\mathsf{P}\top$ and $\geqslant n\,.\mathsf{P}\top$, when they are called *unqualified number restrictions*, or simply *number restrictions*, and are often abbreviated to $\leqslant n\,\mathsf{P}$ and $\geqslant n\mathsf{P}$. The roles that can appear in cardinality restriction concepts are usually restricted to being atomic, as allowing arbitrary roles in such concepts is known no lead to undecidability [Baader and Sattler, 1996b].

Role forming operators may also be supported, and in some very expressive logics roles can be regular expressions formed using *union* (denoted $\sqcup$), *composition* (denoted $\circ$), *reflexive-transitive closure* (denoted *) and *identity* operators (denoted *id*), possibly augmented with the *inverse* (also known as *converse*) operator (denoted $^-$) [De Giacomo and Lenzerini, 1996]. In most implemented systems, however, roles are restricted to being atomic names.

Concepts and roles are given a standard Tarski style model theoretic semantics, their meaning being given by an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, ^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is the domain (a set) and $^{\mathcal{I}}$ is an interpretation function. Full details of both syntax and semantics can be found in Chapter 2.

In general, a DL knowledge base (KB) consists of a set $\mathcal{T}$ of *terminological axioms*, and a set $\mathcal{A}$ of *assertional axioms*. The axioms in $\mathcal{T}$ state facts about concepts and roles while those in $\mathcal{A}$ state facts about individual instances of concepts and roles. As in this chapter we will mostly be concerned with terminological reasoning, that is reasoning about concepts and roles, a KB will usually be taken to consist only of the terminological component $\mathcal{T}$.

A terminological KB $\mathcal{T}$ usually consists of a set of axioms of the form $C \sqsubseteq D$ and $C \equiv D$, where C and D are concepts. An interpretation $\mathcal{I}$ satisfies $\mathcal{T}$ if for every axiom $(C \sqsubseteq D) \in \mathcal{T}$, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and for every axiom $(C \equiv D) \in \mathcal{T}$, $C^{\mathcal{I}} = D^{\mathcal{I}}$; $\mathcal{T}$ is satisfiable if there exists some non empty interpretation that satisfies it. Note that $\mathcal{T}$ can, without loss of generality, be restricted to contain only inclusion axioms or only equality axioms, as the two forms can be reduced one to the other using the following equivalences:

$$
\begin{aligned}
C \sqsubseteq D &\iff \top \equiv D \sqcup \neg C \\
C \equiv D &\iff C \sqsubseteq D \text{ and } D \sqsubseteq C
\end{aligned}
$$

A concept C is *subsumed* by a concept D with respect to $\mathcal{T}$ (written $\mathcal{T} \models C \sqsubseteq D$) if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every interpretation $\mathcal{I}$ that satisfies $\mathcal{T}$, a concept C is *satisfiable* with respect to $\mathcal{T}$ (written $\mathcal{T} \models C \not\sqsubseteq \bot$) if $C^{\mathcal{I}} \neq \emptyset$ in some $\mathcal{I}$ that satisfies $\mathcal{T}$, and a concept C is *unsatisfiable* (not satisfiable) with respect to $\mathcal{T}$ (written $\mathcal{T} \models \neg C$) if $C^{\mathcal{I}} = \emptyset$ in every $\mathcal{I}$ that satisfies $\mathcal{T}$. Subsumption and (un)satisfiability are closely

related. If $\mathcal{T} \models C \sqsubseteq D$, then in all interpretations $\mathcal{I}$ that satisfy $\mathcal{T}$, $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ and so $C^{\mathcal{I}} \cap (\neg D)^{\mathcal{I}} = \emptyset$. Conversely, if C is not satisfiable with respect to $\mathcal{T}$, then in all $\mathcal{I}$ that satisfy $\mathcal{T}$, $C^{\mathcal{I}} = \emptyset$ and so $C^{\mathcal{I}} \subseteq \bot^{\mathcal{I}}$. Subsumption and (un)satisfiability can thus be reduced one to the other using the following equivalences:

$$\mathcal{T} \models C \sqsubseteq D \iff \mathcal{T} \models \neg(C \sqcap \neg D)$$
$$\mathcal{T} \models \neg C \iff \mathcal{T} \models C \sqsubseteq \bot$$

In some DLs $\mathcal{T}$ can also contain axioms that define a set of transitive roles $\mathbf{R}_+$ and/or a subsumption partial ordering on roles [Horrocks and Sattler, 1999]. An axiom $R \in \mathbf{R}_+$ states that R is a transitive role while an axiom $R \sqsubseteq S$ states that R is subsumed by S. An interpretation $\mathcal{I}$ satisfies the axiom $R \in \mathbf{R}_+$ if $R^{\mathcal{I}}$ is transitively closed (i.e, $(R^{\mathcal{I}})^+ = R^{\mathcal{I}}$), and it satisfies the axiom $R \sqsubseteq S$ if $R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$.

9.2.2 Reasoning services

Terminological reasoning in a DL based Knowledge Representation System is based on determining subsumption relationships with respect to the axioms in a KB. As well as answering specific subsumption and satisfiability queries, it is often useful to compute and store (usually in the form of a directed acyclic graph) the subsumption partial ordering of all the concept names appearing in the KB, a procedure known as *classifying* the KB [Patel-Schneider and Swartout, 1993]. Some systems may also be capable of dealing with *assertional* axioms, those concerning instances of concepts and roles, and performing reasoning tasks such as *realisation* (determining the concepts instantiated by a given individual) and *retrieval* (determining the set of individuals that instantiate a given concept) [Baader *et al.*, 1991]. However, we will mostly concentrate on terminological reasoning as it has been more widely used in DL applications. Moreover, given a sufficiently expressive DL, assertional reasoning can be reduced to terminological reasoning [De Giacomo and Lenzerini, 1996].

In practice, many systems use subsumption testing algorithms that are not capable of determining subsumption relationships with respect to an arbitrary KB. Instead, they restrict the kinds of axiom that can appear in the KB so that dependency eliminating substitutions (known as *unfolding*) can be performed prior to evaluating subsumption relationships. These restrictions require that all axioms are *unique, acyclic definitions*. An axiom is called a definition of A if it is of the form $\mathsf{A} \sqsubseteq D$ or $\mathsf{A} \equiv D$, where A is an atomic name, it is unique if the KB contains no other definition of A, and it is acyclic if D does not refer either directly or indirectly (via other axioms) to A. A KB that satisfies these restrictions will be called an *unfoldable* KB.

Definitions of the form $\mathsf{A} \sqsubseteq D$ are sometimes called *primitive* or *necessary*, as

D specifies a necessary condition for instances of A, while those of the form $\mathsf{A} \equiv D$ are sometimes called *non-primitive* or *necessary and sufficient* as D specifies conditions that are both necessary and sufficient for instances of A. In order to distinguish non-definitional axioms, they are often called *general* axioms [Buchheit *et al.*, 1993a]. Restricting the KB to definition axioms makes reasoning much easier, but significantly reduces the expressive power of the DL. However, even with an unrestricted (or *general*) KB, definition axioms and unfolding are still useful ideas, as they can be used to optimise the reasoning procedures (see Section 9.4.3).

9.2.3 Unfolding

Given an unfoldable KB $\mathcal{T}$, and a concept C whose satisfiability is to be tested with respect to $\mathcal{T}$, it is possible to eliminate from C all concept names occurring in $\mathcal{T}$ using a recursive substitution procedure called unfolding. The satisfiability of the resulting concept is independent of the axioms in $\mathcal{T}$ and can therefore be tested using a decision procedure that is only capable of determining the satisfiability of a single concept (or equivalently, the satisfiability of a concept with respect to an empty KB).

For a non-primitive concept name A, defined in $\mathcal{T}$ by an axiom $\mathsf{A} \equiv D$, the procedure is simply to substitute A with D wherever it occurs in C, and then to recursively unfold D. For a primitive concept name A, defined in $\mathcal{T}$ by an axiom $\mathsf{A} \sqsubseteq D$, the procedure is slightly more complex. Wherever A occurs in C it is substituted with the concept $\mathsf{A}' \sqcap D$, where A' is a new concept name not occurring in $\mathcal{T}$ or C, and D is then recursively unfolded. The concept A' represents the "primitiveness" of A—the unspecified characteristics that differentiate it from D. We will use $\mathsf{Unfold}(C, \mathcal{T})$ to denote the concept C unfolded with respect to a KB $\mathcal{T}$.

A decision procedure that tries to find a satisfying interpretation $\mathcal{I}$ for the unfolded concept can now be used, as any such interpretation will also satisfy $\mathcal{T}$. This can easily be shown by applying the unfolding procedure to all of the concepts forming the right hand side of axioms in $\mathcal{T}$, so that they are constructed entirely from concept names that are not defined in $\mathcal{T}$, and are thus independent of the other axioms in $\mathcal{T}$. The interpretation of each defined concept in $\mathcal{T}$ can then be taken to be the interpretation of the unfolded right hand side concept, as given by $\mathcal{I}$ and the semantics of the concept and role forming operators.

Subsumption reasoning can be made independent of $\mathcal{T}$ using the same technique. Given two concepts C and D, determining if C is subsumed by D with respect to $\mathcal{T}$ is the same as determining if $\mathsf{Unfold}(C, \mathcal{T})$ is subsumed by $\mathsf{Unfold}(D, \mathcal{T})$ with

respect to an empty KB:

$$\mathcal{T} \models C \sqsubseteq D \iff \emptyset \models \mathsf{Unfold}(C, \mathcal{T}) \sqsubseteq \mathsf{Unfold}(D, \mathcal{T})$$

Unfolding would not be possible, in general, if the axioms in $\mathcal{T}$ were not unique acyclic definitions. If $\mathcal{T}$ contained multiple definition axioms for some concept A, for example $\{(\mathsf{A} \equiv C), (\mathsf{A} \equiv D)\} \subseteq \mathcal{T}$, then it would not be possible to make a substitution for A that preserved the meaning of both axioms. If $\mathcal{T}$ contained cyclical axioms, for example $(\mathsf{A} \sqsubseteq \exists R.\mathsf{A}) \in \mathcal{T}$, then trying to unfold A would lead to non-termination. If $\mathcal{T}$ contained general axioms, for example $\exists R.C \sqsubseteq D$, then it could not be guaranteed that an interpretation satisfying the unfolded concept would also satisfy these axioms.

9.2.4 Internalisation

While it is possible to design an algorithm capable of reasoning with respect to a general KB [Buchheit *et al.*, 1993a], with more expressive logics, in particular those allowing the definition of a *universal role*, a procedure called *internalisation* can be used to reduce the problem to that of determining the satisfiability of a single concept [Baader, 1991]. A truly universal role is one whose interpretation includes every pair of elements in the domain of interpretation (i.e., $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$). However, a role U is universal w.r.t. a terminology $\mathcal{T}$ if it is defined such that U is transitively closed and $\mathsf{P} \sqsubseteq U$ for all role names P occurring in $\mathcal{T}$. For a logic that supports the union and transitive reflexive closure role forming operators, this can be achieved simply by taking U to be

$$(\mathsf{P}_1 \sqcup \ldots \sqcup \mathsf{P}_n \sqcup \mathsf{P}_1^- \sqcup \ldots \sqcup \mathsf{P}_n^-)^*,$$

where $\mathsf{P}_1, \ldots, \mathsf{P}_n$ are all the roles names occurring in $\mathcal{T}$. For a logic that supports transitively closed roles and role inclusion axioms, this can be achieved by adding the axioms

$$(U \in \mathbf{R}_+), (\mathsf{P}_1 \sqsubseteq U), \ldots, (\mathsf{P}_n \sqsubseteq U), (\mathsf{P}_1^- \sqsubseteq U), \ldots, (\mathsf{P}_n^- \sqsubseteq U)$$

to $\mathcal{T}$, where $\mathsf{P}_1, \ldots, \mathsf{P}_n$ are all the roles names occurring in $\mathcal{T}$ and U is a new role name not occurring in $\mathcal{T}$. Note that in either case, the inverse role components are only required if the logic supports the inverse role operator.

The concept axioms in $\mathcal{T}$ can be reduced to axioms of the form $\top \sqsubseteq C$ using the equivalences:

$$A \equiv B \iff \top \sqsubseteq (A \sqcup \neg B) \sqcap (\neg A \sqcup B)$$
$$A \sqsubseteq B \iff \top \sqsubseteq \neg A \sqcup B$$

These axioms can then be conjoined to give a single axiom $\top \sqsubseteq C$, where

$$C = \bigsqcap_{(A_i \equiv B_i) \in \mathcal{T}} ((A_i \sqcup \neg B_i) \sqcap (\neg A_i \sqcup B_i)) \sqcap \bigsqcap_{(A_j \sqsubseteq B_j) \in \mathcal{T}} (\neg A_j \sqcup B_j)$$

Because the interpretation of $\top$ is equal to the domain ($\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$), this axiom states that every element in the domain must satisfy C. When testing the satisfiability of a concept D with respect to $\mathcal{T}$, this constraint on possible interpretations can be imposed by testing the satisfiability of $D \sqcap C \sqcap \forall U.C$ (or simply $D \sqcap \forall U.C$ in the case where U is transitively reflexively closed). This relies on the fact that satisfiable DL concepts always have an interpretation in which every element is connected to every other element by some sequence of roles (the collapsed model property) [Schild, 1991].

9.3 Subsumption testing algorithms

The use of unfolding and internalisation means that, in most cases, terminological reasoning in a Description Logic based Knowledge Representation System can be reduced to subsumption or satisfiability reasoning. There are several algorithmic techniques for computing subsumption relationships, but they divide into two main families: structural and logical.

9.3.1 Structural subsumption algorithms

Structural algorithms were used in early DL system such as Kl-One [Brachman and Schmolze, 1985], Nikl [Kaczmarek *et al.*, 1986] and Krypton [Brachman *et al.*, 1983a], and are still used in systems such as Classic [Patel-Schneider *et al.*, 1991], Loom [MacGregor, 1991b] and Grail [Rector *et al.*, 1997]. To determine if one concept subsumes another, structural algorithms simply compare the (normalised) syntactic structure of the two concepts (see Chapter 2).

Although such algorithms can by quite efficient [Borgida and Patel-Schneider, 1994; Heinsohn *et al.*, 1994], they have several disadvantages.

- Perhaps the most important disadvantage of this type of algorithm is that while it is generally easy to demonstrate the soundness of the structural inference rules (they will never infer an invalid subsumption relationship), they are usually incomplete (they may fail to infer all valid subsumption relationships).
- It is difficult to extend structural algorithms in order to to deal with more expressive logics, in particular those supporting general negation, or to reason with respect to an arbitrary KB. This lack of expressive power makes the DL system of limited value in traditional ontological engineering applications [Doyle and

Patil, 1991], and completely useless in DataBase schema reasoning applications [Calvanese *et al.*, 1998f].

- Although accepting some degree of incompleteness is one way of improving the performance of a DL reasoner, the performance of incomplete reasoners is highly dependent on the degree of incompleteness, and this is notoriously difficult to quantify [Borgida, 1992a].

9.3.2 Logical algorithms

These kinds of algorithm use a refutation style proof: C is subsumed by D if it can be shown that the existence of an individual x that is in the extension of C ($x \in C^{\mathcal{I}}$) but not in the extension of D ($x \notin D^{\mathcal{I}}$) is logically inconsistent. As we have seen in Section 9.2.2, this corresponds to testing the logical (un)satisfiability of the concept $C \sqcap \neg D$ (i.e., $C \sqsubseteq D$ iff $C \sqcap \neg D$ is not satisfiable). Note that forming this concept obviously relies on having full negation in the logic.

Various techniques can be used to test the logical satisfiability of a concept. One obvious possibility is to exploit an existing reasoner. For example, the LogicsWorkbench [Balsiger *et al.*, 1996], a general purpouse proposition modal logic reasoning system, could be used simply by exploiting the well known correspondences between description and modal logics [Schild, 1991]. First order logic theorem provers can also be used via appropriate traslations of DLs into first order logic. Examples of this approach can be seen in systems developed by Hustadt and Schmidt [1997], using the Spass theorem prover, and Paramasivam and Plaisted [1998], using the CLIN-S theorem prover. An existing reasoner could also be used as a component of a more powerful system, as in Ksat/*Sat [Giunchiglia and Sebastiani, 1996a; Giunchiglia *et al.*, 2001a], where a propositional satisfiability (SAT) tester is used as the key component of a propositional modal satisfiability reasoner.

There are advantages and disadvantages to the "re-use" approach. On the positive side, it should be much easier to build a system based on an existing reasoner, and performance can be maximised by using a state of the art implementation such as Spass (a highly optimised first order theorem prover) or the highly optimised SAT testing algorithms used in Ksat and *Sat (the use of a specialised SAT tester allows *Sat to outperform other systems on classes of problem that emphasise propositional reasoning). The translation (into first order logic) approach has also been shown to be able to deal with a wide range of expressive DLs, in particular those with complex role forming operators such as negation or identity [Hustadt and Schmidt, 2000].

On the negative side, it may be difficult to extend the reasoner to deal with more expressive logics, or to add optimisations that take advantage of specific features

of the DL, without reimplementing the reasoner (as has been done, for example, in more recent versions of the *Sat system).

Most, if not all, implemented DL systems based on logical reasoning have used custom designed tableaux decision procedures. These algorithms try to prove that D subsumes C by starting with a single individual satisfying $C \sqcap \neg D$, and demonstrating that any attempt to extend this into a complete interpretation (using a set of *tableaux expansion rules*) will lead to a logical contradiction. If a complete and non-contradictory interpretation is found, then this represents a counter example (an interpretation in which some element of the domain is in $C^{\mathcal{I}}$ but not in $D^{\mathcal{I}}$) that disproves the conjectured subsumption relationship.

This approach has many advantages and has dominated recent DL research:

- it has a sound theoretical basis in first order logic [Hollunder *et al.*, 1990];
- it can be relatively easily adapted to allow for a range of logical languages by changing the set of tableaux expansion rules [Hollunder *et al.*, 1990; Bresciani *et al.*, 1995];
- it can be adapted to deal with very expressive logics, and to reason with respect to an arbitrary KB, by using more sophisticated control mechanisms to ensure termination [Baader, 1991; Buchheit *et al.*, 1993c; Sattler, 1996];
- it has been shown to be optimal for a number of DL languages, in the sense that the worst case complexity of the algorithm is no worse than the known complexity of the satisfiability problem for the logic [Hollunder *et al.*, 1990].

In the remainder of this chapter, detailed descriptions of implementation and optimisation techniques will assume the use of a tableaux decision procedure. However, many of the techniques are independent of the subsumption testing algorithm or could easily be adapted to most logic based methods. The reverse is also true, and several of the described techniques have been adapted from other logical decision procedures, in particular those that try to optimise the search used to deal with non-determinism.

9.3.2.1 Tableaux algorithms

Tableaux algorithms try to prove the satisfiability of a concept D by constructing a *model*, an interpretation $\mathcal{I}$ in which $D^{\mathcal{I}}$ is not empty. A *tableau* is a graph which represents such a model, with nodes corresponding to individuals (elements of $\Delta^{\mathcal{I}}$) and edges corresponding to relationships between individuals (elements of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$).

A typical algorithm will start with a single individual satisfying D and try to construct a tableau, or some structure from which a tableau can be constructed, by inferring the existence of additional individuals or of additional constraints on individuals. The inference mechanism consists of applying a set of expansion rules

which correspond to the logical constructs of the language, and the algorithm terminates either when the structure is complete (no further inferences are possible) or obvious contradictions have been revealed. Non-determinism is dealt with by searching different possible expansions: the concept is unsatisfiable if every expansion leads to a contradiction and is satisfiable if any possible expansion leads to the discovery of a complete non-contradictory structure.

Theoretical presentations of tableaux algorithms use a variety of notational styles including constraints [Hollunder *et al.*, 1990], prefixes [De Giacomo and Massacci, 1996] and labelled graphs [Sattler, 1996]. We will use the labelled graph notation as it has an obvious correspondence with standard implementation techniques. In its basic form, this notation describes the construction of a directed graph (usually a tree) in which each node x is labelled with a set of concepts ($\mathcal{L}(x) = \{C_1, \ldots, C_n\}$), and each edge $\langle x, y \rangle$ is labelled with a role ($\mathcal{L}(\langle x, y \rangle) = R$). When a concept C is in the label of a node x ($C \in \mathcal{L}(x)$), it represents a model in which the individual corresponding with x is in the interpretation of C. When an edge $\langle x, y \rangle$ is labelled R ($\mathcal{L}(\langle x, y \rangle) = R$), it represents a model in which the tuple corresponding with $\langle x, y \rangle$ is in the interpretation of R. A node y is called an R-successor of a node x if there is an edge $\langle x, y \rangle$ labelled R, x is called the predecessor of y if y is an R-successor of x, and x is called an ancestor of y if x is the predecessor of y or there exists some node z such that z is the predecessor of y and x is an ancestor of z. A contradiction or *clash* is detected when $\{C, \neg C\} \subseteq \mathcal{L}(x)$ for some concept C and some node x.

To test the satisfiability of a concept D, a basic algorithm initialises a tree to contain a single node x (called the *root* node) with $\mathcal{L}(x) = \{D\}$, and then expands the tree by applying rules that either extend node labels or add new leaf nodes. A set of expansion rules for the $\mathcal{ALC}$ description logic is shown in Figure 9.1, where C and D are concepts, and R is a role. Note that:

- Concepts are assumed to be in *negation normal form*, that is with negations only applying to concept names. Arbitrary $\mathcal{ALC}$ concepts can be converted to negation normal form by pushing negations inwards using a combination of DeMorgan's laws and the equivalences $\neg(\exists R.C) \iff (\forall R.\neg C)$ and $\neg(\forall R.C) \iff (\exists R.\neg C)$. This procedure can be extended to more expressive logics using additional equivalences such as $\neg(\leqslant n\,\mathsf{R}) \iff (\geqslant (n+1)R)$.
- Disjunctive concepts $(C \sqcup D) \in \mathcal{L}(x)$ give rise to non-deterministic expansion. In practice this is usually dealt with by search: trying each possible expansion in turn until a fully expanded and clash free tree is found, or all possibilities have been shown to lead to contradictions. In more expressive logics other constructs, such as maximum number restrictions $(\leqslant n\,\mathsf{R})$, also lead to non-deterministic expansion. Searching non-deterministic expansions is the main cause of intractability in tableaux subsumption testing algorithms.

$\sqcap$-rule	if 1.	$(C \sqcap D) \in \mathcal{L}(x)$
	2.	$\{C, D\} \not\subseteq \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C, D\}$
$\sqcup$-rule	if 1.	$(C \sqcup D) \in \mathcal{L}(x)$
	2.	$\{C, D\} \cap \mathcal{L}(x) = \emptyset$
	then	*either* $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$
		or $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{D\}$
$\exists$-rule	if 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \in \mathcal{L}(y)$
	then	create a new node y and edge $\langle x, y \rangle$
		with $\mathcal{L}(y) = \{C\}$ and $\mathcal{L}(\langle x, y \rangle) = R$
$\forall$-rule	if 1.	$\forall R.C \in \mathcal{L}(x)$
	2.	there is some y s.t. $\mathcal{L}(\langle x, y \rangle) = R$ and $C \notin \mathcal{L}(y)$
	then	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Fig. 9.1. Tableaux expansion rules for $\mathcal{ALC}$.

- Existential role restriction concepts $\exists R.C \in \mathcal{L}(x)$ cause the creation of new R-successor nodes, and universal role restriction concepts $\forall R.C \in \mathcal{L}(x)$ extend the labels of R-successor nodes.

The tree is fully expanded when none of the expansion rules can be applied. If a fully expanded and clash-free tree can be found, then the algorithm returns *satisfiable*; otherwise it returns *unsatisfiable*.

More expressive logics may require several extensions to this basic formalism. For example, with logics that include both role inclusion axioms and some form of cardinality restriction, it may be necessary to label edges with sets of role names instead of a single role name [Horrocks, 1998b]. It may also be necessary to add cycle detection (often called *blocking*) to the preconditions of some of the inference rules in order to guarantee termination [Buchheit *et al.*, 1993a; Baader *et al.*, 1996], the general idea being to stop the expansion of a branch whenever the same node label recurs in the branch. Blocking can also lead to a more complex correspondence between the structure created by the algorithm and a model of a satisfiable concept, as the model may contain cycles or even be non-finite [Horrocks and Sattler, 1999].

9.4 Theory versus practice

So far, what we have seen is typical of theoretical presentations of tableaux based decision procedures. Such a presentation is sufficient for soundness and completeness proofs, and is an essential starting point for the implementation of a reliable subsumption testing algorithm. However, there often remains a considerable gap between the theoretical algorithm and an actual implementation. Additional points which may need to be considered are:

- the efficiency of the algorithm, in the theoretical (worst case) sense;
- the efficiency of the algorithm, in a practical (typical case) sense;
- how to use the algorithm for reasoning with unfoldable, general and cyclical KBs;
- optimising the (implementation of the) algorithm to improve the typical case performance.

In the remainder of this Section we will consider the first three points, while in the following section we will consider implementation and optimisation techniques in detail.

9.4.1 Worst case complexity

When considering an implementation, it is sensible to start with an algorithm that is known to be theoretically efficient, even if the implementation subsequently departs from the theory to some extent. Theoretically efficient is taken to mean that the complexity of the algorithm is equal to the complexity of the satisfiability problem for the logic, where this is known, or at least that consideration has been given to the worst case complexity of the algorithm. This is not always the case, as the algorithm may have been designed to facilitate a soundness and completeness proof, with little consideration having been given to worst case complexity, much less implementation.

Apart from establishing an upper bound for the "hardness" of the problem, studies of theoretical complexity can suggest useful implementation techniques. For example, a study of the complexity of the satisfiability problem for $\mathcal{ALC}$ concepts with respect to a general KB has demonstrated that caching of intermediate results is required in order to stay in ExpTime [Donini *et al.*, 1996a], while studying the complexity of the satisfiability problem for $\mathcal{SIN}$ concepts has shown that a more sophisticated labelling and blocking strategy can be used in order to stay in PSpace [Horrocks *et al.*, 1999].

One theoretically derived technique that is widely used in practice is the *trace* technique. This is a method for minimising the amount of space used by the algorithm to store the tableaux expansion tree. The idea is to impose an ordering on the application of expansion rules so that local *propositional reasoning* (finding a clash-free expansion of conjunctions and disjunctions using the $\sqcap$-rule and $\sqcup$-rule) is completed before new nodes are created using the $\exists$-rule. A successor created by an application of the $\exists$-rule, and any possible applications of the $\forall$-rule, can then be treated as an independent sub-problem that returns either *satisfiable* or *unsatisfiable*, and the space used to solve it can be reused in solving the next sub-problem. A node x returns *satisfiable* if there is a clash-free propositional solution for which any and all sub-problems return *satisfiable*; otherwise it returns *unsatisfiable*. In

$\exists\forall$-rule	if 1.	$\exists R.C \in \mathcal{L}(x)$
	2.	there is no y s.t. $\mathcal{L}(\langle x, y\rangle) = R$ and $C \in \mathcal{L}(y)$
	3.	neither the $\sqcap$-rule nor the $\sqcup$-rule is applicable to $\mathcal{L}(x)$
	then	create a new node y and edge $\langle x, y\rangle$
		with $\mathcal{L}(y) = \{C\} \cup \{D \mid \forall R.D \in \mathcal{L}(x)\}$ and $\mathcal{L}(\langle x, y\rangle) = R$

Fig. 9.2. Combined $\exists\forall$-rule for $\mathcal{ALC}$.

algorithms where the trace technique can be used, the $\forall$-rule is often incorporated in the $\exists$-rule, giving a single rule as shown in Figure 9.2.

Apart from minimising space usage the trace technique is generally viewed as a sensible way of organising the expansion and the flow of control within the algorithm. Ordering the expansion in this way may also be required by some blocking strategies [Buchheit *et al.*, 1993a], although in some cases it is possible to use a more efficient subset blocking technique that is independent of the ordering [Baader *et al.*, 1996].

The trace technique relies on the fact that node labels are not affected by the expansion of their successors. This is no longer true when the logic includes inverse roles, because universal value restrictions in the label of a successor of a node x can augment $\mathcal{L}(x)$. This could invalidate the existing propositional solution for $\mathcal{L}(x)$, or invalidate previously computed solutions to sub-problems in other successor nodes. For example, if

$$\mathcal{L}(x) = \{\exists R.C, \exists S.(\forall S^{-}.(\forall R.\neg C))\},$$

then x is obviously unsatisfiable as expanding $\exists S.(\forall S^{-}.(\forall R.\neg C))$ will add $\forall R.\neg C$ to $\mathcal{L}(x)$, meaning that x must have an R-successor whose label contains both C and $\neg C$. The contradiction would not be discovered if the R-successor required by $\exists R.C \in \mathcal{L}(x)$ were generated first, found to be satisfiable and then deleted from the tree in order to save space.

The development of a PSpace algorithm for the $\mathcal{SIN}$ logic has shown that a modified version of the trace technique can still be used with logics that include inverse roles [Horrocks *et al.*, 1999]. However, the modification requires that the propositional solution and all sub-problems are re-computed whenever the label of a node is augmented by the expansion of a universal value restriction in the label of one of its successors.

9.4.2 Typical case complexity

Although useful practical techniques can be derived from the study of theoretical algorithms, it should be borne in mind that minimising worst case complexity may require the use of techniques that clearly would not be sensible in typical cases. This is because the kinds of pathological problem that would lead to worst case

behaviour do not seem to occur in realistic applications. In particular, the amount of space used by algorithms does not seem to be a practical problem, whereas the time taken for the computation certainly is. For example, in experiments with the FaCT system using the DL'98 test suite, available memory (200Mb) was never exhausted in spite of the fact that some single computations required hundreds of seconds of CPU time [Horrocks and Patel-Schneider, 1998b]. In other experiments using the Galen KB, computations were run for tens of thousands of seconds of CPU time without exhausting available memory.

In view of these considerations, techniques that save space by recomputing are unlikely to be of practical value. The modified trace technique used in the PSpace $\mathcal{SIN}$ algorithm (see Section 9.4.1), for example, is probably not of practical value. However, the more sophisticated labelling and blocking strategy, which allows the establishment a polynomial bound on the length of branches, could be used not only in an implementation of the $\mathcal{SIN}$ algorithm, but also in implementations of more expressive logics where other considerations mean that the PSpace result no longer holds [Horrocks *et al.*, 1999].

In practice, the poor performance of tableaux algorithms is due to non-determinism in the expansion rules (for example the $\sqcup$-rule), and the resulting search of different possible expansions. This is often treated in a very cursory manner in theoretical presentations. For soundness and completeness it is enough to prove that the search will always find a solution if one exists, and that it will always terminate. For worst case complexity, an upper bound on the size of the search space is all that is required. As this upper bound is invariably exponential with respect to the size of the problem, exploring the whole search space would inevitably lead to intractability for all but the smallest problems. When implementing an algorithm it is therefore vital to give much more careful consideration to non-deterministic expansion, in particular how to reduce the size of the search space and how to explore it in an efficient manner. Many of the optimisations discussed in subsequent sections will be aimed at doing this, for example by using *absorption* to localise non-determinism in the KB, *dependency directed backtracking* to prune the search tree, *heuristics* to guide the search, and *caching* to avoid repetitive search.

9.4.3 Reasoning with a knowledge base

One area in which the theory and practice diverge significantly is that of reasoning with respect to the axioms in a KB. This problem is rarely considered in detail: with less expressive logics the KB is usually restricted to being unfoldable, while with more expressive logics, all axioms can be treated as general axioms and dealt with via internalisation. In either case it is sufficient to consider an algorithm that tests the satisfiability of a single concept, usually in negation normal form.

In practice, it is much more efficient to retain the structure of the KB for as long as possible, and to take advantage of it during subsumption/satisfiability testing. One way in which this can be done is to use *lazy unfolding*—only unfolding concepts as required by the progress of the subsumption or satisfiability testing algorithm [Baader *et al.*, 1992a]. With a tableaux algorithm, this means that a defined concept A is only unfolded when it occurs in a node label. For example, if $\mathcal{T}$ contains the non-primitive definition axiom $\mathsf{A} \equiv C$, and the $\sqcap$-rule is applied to a concept $(\mathsf{A} \sqcap D) \in \mathcal{L}(x)$ so that A and D are added to $\mathcal{L}(x)$, then at this point A can be unfolded by substituting it with C.

Used in this way, lazy unfolding already has the advantage that it avoids unnecessary unfolding of irrelevant sub-concepts, either because a contradiction is discovered without fully expanding the tree, or because a non-deterministic expansion choice leads to a complete and clash free tree. However, a much greater increase in efficiency can be achieved if, instead of substituting concept names with their definitions, names are retained when their definitions are added. This is because the discovery of a clash between concept names can avoid expansion of their definitions [Baader *et al.*, 1992a].

In general, lazy unfolding can be described as additional tableaux expansion rules, defined as follows.

U_1-rule	if 1.	$\mathsf{A} \in \mathcal{L}(x)$ and $(\mathsf{A} \equiv C) \in \mathcal{T}$
	2.	$C \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$
U_2-rule	if 1.	$\neg\mathsf{A} \in \mathcal{L}(x)$ and $(\mathsf{A} \equiv C) \in \mathcal{T}$
	2.	$\neg C \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{\neg C\}$
U_3-rule	if 1.	$\mathsf{A} \in \mathcal{L}(x)$ and $(\mathsf{A} \sqsubseteq C) \in \mathcal{T}$
	2.	$C \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$

The U_1-rule and U_2-rule reflect the symmetry of the equality relation in the non-primitive definition $\mathsf{A} \equiv C$, which is equivalent to $\mathsf{A} \sqsubseteq C$ and $\neg\mathsf{A} \sqsubseteq \neg C$. The U_3-rule on the other hand reflects the asymmetry of the subsumption relation in the primitive definition $\mathsf{A} \sqsubseteq C$.

Treating all the axioms in the KB as general axioms, and dealing with them via internalisation, is also highly inefficient. For example, if $\mathcal{T}$ contains an axiom $\mathsf{A} \sqsubseteq C$, where A is a concept name not appearing on the left hand side of any other axiom, then it is easy to deal with the axiom using the lazy unfolding technique, simply adding C to the label of any node in which A appears. Treating all axioms

as general axioms would be equivalent to applying the following additional tableaux expansion rules:

I_1-rule	if 1.	$(C \equiv D) \in \mathcal{T}$
	2.	$(D \sqcup \neg C) \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{(D \sqcup \neg C)\}$
I_2-rule	if 1.	$(C \equiv D) \in \mathcal{T}$
	2.	$(\neg D \sqcup C) \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{(\neg D \sqcup C)\}$
I_3-rule	if 1.	$(C \sqsubseteq D) \in \mathcal{T}$
	2.	$(D \sqcup \neg C) \notin \mathcal{L}(x)$
	then	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{(D \sqcup \neg C)\}$

With $(\mathsf{A} \sqsubseteq C) \in \mathcal{T}$, this would result in the disjunction $(C \sqcup \neg \mathsf{A})$ being added to the label of every node, leading to non-deterministic expansion and search, the main cause of empirical intractability.

The solution to this problem is to divide the KB into two components, an unfoldable part $\mathcal{T}_u$ and a general part $\mathcal{T}_g$, such that $\mathcal{T}_g = \mathcal{T} \setminus \mathcal{T}_u$, and $\mathcal{T}_u$ contains unique, acyclical, definition axioms. This is easily achieved, e.g., by initialising $\mathcal{T}_u$ to $\emptyset$ (which is obviously unfoldable), then for each axiom X in $\mathcal{T}$, adding X to $\mathcal{T}_u$ if $\mathcal{T}_u \cup X$ is still unfoldable, and adding X to $\mathcal{T}_g$ otherwise.[1] It is then possible to use lazy unfolding to deal with $\mathcal{T}_u$, and internalisation to deal with $\mathcal{T}_g$.

Given that the satisfiability testing algorithm includes some sort of cycle checking, such as blocking, then it is even possible to be a little less conservative with respect to the definition of $\mathcal{T}_u$ by allowing it to contain cyclical primitive definition axioms, for example axioms of the form $\mathsf{A} \sqsubseteq \exists R.\mathsf{A}$. Lazy unfolding will ensure that $\mathsf{A}^{\mathcal{I}} \subseteq \exists R.\mathsf{A}^{\mathcal{I}}$ by adding $\exists R.\mathsf{A}$ to every node containing A, while blocking will take care of the non-termination problem that such an axiom would otherwise cause [Horrocks, 1997b]. Moreover, multiple primitive definitions for a single name can be added to $\mathcal{T}_u$, or equivalently merged into a single definition using the equivalence

$$(\mathsf{A} \sqsubseteq C_1), \ldots, (\mathsf{A} \sqsubseteq C_n) \iff \mathsf{A} \sqsubseteq (C_1 \sqcap \ldots \sqcap C_n)$$

However, if $\mathcal{T}_u$ contains a non-primitive definition axiom $\mathsf{A} \equiv C$, then it cannot contain any other definitions for A, because this would be equivalent to allowing general axioms in $\mathcal{T}_u$. For example, given a general axiom $C \sqsubseteq D$, this could be added to $\mathcal{T}_u$ as $\mathsf{A} \sqsubseteq D$ and $\mathsf{A} \equiv C$, where A is a new name not appearing in $\mathcal{T}$. Moreover, certain kinds of non-primitive cycles cannot be allowed as they can be used to constrain possible models a way that would not be reflected by unfolding. For example, if $(\mathsf{A} \equiv \neg \mathsf{A}) \in \mathcal{T}$ for some concept name A, then the domain of all

[1] Note that the result may depend on the order in which the axioms in $\mathcal{T}$ are processed.

valid interpretations of $\mathcal{T}$ must be empty, and $\mathcal{T} \models C \sqsubseteq D$ for all concepts C and D [Horrocks and Tobies, 2000].

9.5 Optimisation techniques

The Kris system demonstrated that by taking a well designed tableaux algorithm, and applying some reasonable implementation and optimisation techniques (such as lazy expansion), it is possible to obtain a tableaux based DL system that behaves reasonably well in typical cases, and compares favourably with systems based on structural algorithms [Baader *et al.*, 1992a]. However, this kind of system is still much too slow to be usable in many realistic applications. Fortunately, it is possible to achieve dramatic improvements in typical case performance by using a wider range of optimisation techniques.

As DL systems are often used to classify a KB, a hierarchy of optimisation techniques is naturally suggested based on the stage of the classification process at which they can be applied.

(i) Preprocessing optimisations that try to modify the KB so that classification and subsumption testing are easier.
(ii) Partial ordering optimisations that try to minimise the number of subsumption tests required in order to classify the KB
(iii) Subsumption optimisations that try to avoid performing a potentially expensive satisfiability test, usually by substituting a cheaper test.
(iv) Satisfiability optimisations that try to improve the typical case performance of the underlying satisfiability tester.

9.5.1 Preprocessing optimisations

The axioms that constitute a DL KB may have been generated by a human knowledge engineer, as is typically the case in ontological engineering applications, or be the result of some automated mapping from another formalism, as is typically the case in DB schema and query reasoning applications. In either case it is unlikely that a great deal of consideration was given to facilitating the subsequent reasoning procedures; the KB may, for example, contain considerable redundancy and may make unnecessary use of general axioms. As we have seen, general axioms are costly to reason with due to the high degree of non-determinism that they introduce.

It is, therefore, useful to preprocess the KB, applying a range of syntactic simplifications and manipulations. The first of these, *normalisation*, tries to simplify the KB by identifying syntactic equivalences, contradictions and tautologies. The second, *absorption*, tries to eliminate general axioms by augmenting definition axioms.

9.5.1.1 Normalisation

In realistic KBs, at least those manually constructed, large and complex concepts are seldom described monolithically, but are built up from a hierarchy of named concepts whose descriptions are less complex. The lazy unfolding technique described above can use this structure to provide more rapid detection of contradictions.

The effectiveness of lazy unfolding is greatly increased if a contradiction between two concepts can be detected whenever one is syntactically equivalent to the negation of the other; for example, we would like to discover a direct contradiction between $(C \sqcap D)$ and $(\neg D \sqcup \neg C)$. This can be achieved by transforming all concepts into a syntactic normal form, and by directly detecting contradictions caused by non-atomic concepts as well as those caused by concept names.

In DLs there is often redundancy in the set of concept forming operators. In particular, logics with full negation often provide pairs of operators, either one of which can be eliminated in favour of the other by using negation. Conjunction and disjunction operators are an example of such a pair, and one can be eliminated in favour of the other using DeMorgan's laws. In syntactic normal form, all concepts are transformed so that only one of each such pair appears in the KB (it does not matter which of the two is chosen, the important thing is uniformity). In $\mathcal{ALC}$, for example, all concepts could be transformed into (possibly negated) value restrictions, conjunctions and atomic concept names, with $(\neg D \sqcup \neg C)$ being transformed into $\neg(D \sqcap C)$. An important refinement is to treat conjunctions as sets (written $\sqcap\{C_1, \ldots, C_n\}$) so that reordering or repeating the conjuncts does not effect equivalence; for example, $(D \sqcap C)$ would be normalised as $\sqcap\{C, D\}$.[1] Normalisation can also include a range of simplifications so that syntactically obvious contradictions and tautologies are detected; for example, $\exists R.\bot$ could be simplified to $\bot$.

Figure 9.3 describes normalisation and simplification functions Norm and Simp for $\mathcal{ALC}$. These can be extended to deal with more expressive logics by adding appropriate normalisations (and possibly additional simplifications). For example, number restrictions can be dealt with by adding the normalisations $\mathsf{Norm}(\leqslant n\, \mathsf{R}) = \neg\!\geqslant (n+1)R$ and $\mathsf{Norm}(\geqslant nR) = \,\geqslant nR$, and the simplification $\mathsf{Simp}(\geqslant 0R) = \top$.

Normalised and simplified concepts may not be in negation normal form, but they can be dealt with by treating them exactly like their non-negated counterparts. For example, $\neg\sqcap\{C, D\}$ can be treated as $(\neg C \sqcup \neg D)$ and $\neg\forall R.C$ can be treated as $\exists R.\neg C$. In the remainder of this chapter we will use both forms interchangeably, choosing whichever is most convenient.

Additional simplifications would clearly be possible. For example, $\forall R.C \sqcap \forall R.D$ could be simplified to $\forall R.\,\mathsf{Norm}(C \sqcap D)$. Which simplifications it is sensible to perform is an implementation decision that may depend on a cost-benefit analysis

[1] Sorting the elements in conjuctions, and eliminating duplicates, achieves the same result.

$$
\begin{array}{rcl}
\mathsf{Norm}(\mathsf{A}) & = & \mathsf{A} \quad \text{for atomic concept name } \mathsf{A} \\
\mathsf{Norm}(\neg C) & = & \mathsf{Simp}(\neg\, \mathsf{Norm}(C)) \\
\mathsf{Norm}(C_1 \sqcap \ldots \sqcap C_n) & = & \mathsf{Simp}(\sqcap\{\mathsf{Norm}(C_1)\} \cup \ldots \cup \{\mathsf{Norm}(C_n)\}) \\
\mathsf{Norm}(C_1 \sqcup \ldots \sqcup C_n) & = & \mathsf{Norm}(\neg(\neg C_1 \sqcap \ldots \sqcap \neg C_n)) \\
\mathsf{Norm}(\forall R.C) & = & \mathsf{Simp}(\forall R.\, \mathsf{Norm}(C)) \\
\mathsf{Norm}(\exists R.C) & = & \mathsf{Norm}(\neg \forall R. \neg C) \\
\\
\mathsf{Simp}(\mathsf{A}) & = & \mathsf{A} \text{ for atomic concept name } \mathsf{A} \\
\mathsf{Simp}(\neg C) & = & \begin{cases} \bot & \text{if } C = \top \\ \top & \text{if } C = \bot \\ \mathsf{Simp}(D) & \text{if } C = \neg D \\ \neg C & \text{otherwise} \end{cases} \\
\mathsf{Simp}(\sqcap \mathbf{S}) & = & \begin{cases} \bot & \text{if } \bot \in \mathbf{S} \\ \bot & \text{if } \{C, \neg C\} \subseteq \mathbf{S} \\ \top & \text{if } \mathbf{S} = \emptyset \\ \mathsf{Simp}(\mathbf{S} \setminus \{\top\}) & \text{if } \top \in \mathbf{S} \\ \mathsf{Simp}(\sqcap \mathbf{P} \cup \mathbf{S} \setminus \{\sqcap\{\mathbf{P}\}\}) & \text{if } \sqcap\{\mathbf{P}\} \in \mathbf{S} \\ \sqcap \mathbf{S} & \text{otherwise} \end{cases} \\
\mathsf{Simp}(\forall R.C) & = & \begin{cases} \top & \text{if } C = \top \\ \forall R.C & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 9.3. Normalisation and simplification functions for $\mathcal{ALC}$.

with respect to some particular application. Empirically, simplification seems to be more effective with mechanically generated KBs and satisfiability problems, in particular those where there the number of different roles is very small. With this kind of problem it is quite common for satisfiability tests to be greatly simplified, or even completely avoided, by simplifying part or all of the concept to either $\top$ or $\bot$. In the benchmark tests used for the Tableaux'98 comparison of modal logic theorem provers, for example, some classes of problem can be completely solved via this mechanism [Heuerding and Schwendimann, 1996; Balsiger and Heuerding, 1998].

If the subsumption testing algorithm is to derive maximum benefit from normalisation, it is important that it directly detect contradictions caused by non-atomic concepts as well as those caused by concept names; for example the occurrence of both $\sqcap\{C, D\}$ and $\neg\sqcap\{C, D\}$ in a node label should be detected as a contradiction without the need for further expansion. This can be achieved by replacing all equivalent (identically encoded) non-atomic concepts C in the KB with a new atomic concept name A, and adding the axiom $\mathsf{A} \equiv C$ to the KB. For example, all occurrences of $\sqcap\{C, D\}$ in a KB could be replaced with CD, and the axiom $\mathsf{CD} \equiv \sqcap\{C, D\}$ added to the KB.

It is necessary to distinguish these newly introduced *system* names from *user* names appearing in the original KB, as system names need not be classified (indeed, it would be very confusing for the user if they were). In practice, it is often more convenient to avoid this problem by using pointer or object identifiers to refer to concepts, with the same identifier always being associated with equivalent concepts. A contradiction is then detected whenever a pointer/identifier and its negation occur in a node label.

The advantages of the normalisation and simplification procedure are:

- It is easy to implement and could be used with most logics and algorithms.
- Subsumption/satisfiability problems can often be simplified, and sometimes even completely avoided, by detecting syntactically obvious satisfiability and unsatisfiability.
- It complements lazy unfolding and improves early clash detection.
- The elimination of redundancies and the sharing of syntactically equivalent structures may lead to the KB being more compactly stored.

The disadvantages are:

- The overhead involved in the procedure, although this is relatively small.
- For very unstructured KBs there may be no benefit, and it might even slightly increase size of KB.

9.5.1.2 Absorption

As we have seen in Section 9.4.3, general axioms are costly to reason with due to the high degree of non-determinism that they introduce. With a tableaux algorithm, a disjunction is added to the label of each node for each general axiom in the KB. This leads to an exponential increase in the search space as the number of nodes and axioms increases. For example, with 10 nodes and a KB containing 10 general axioms there are already 100 disjunctions, and they can be non-deterministically expanded in 2^{100} different ways. For a KB containing large numbers of general axioms (there are 1,214 in the Galen medical terminology KB) this can degrade performance to the extent that subsumption testing is effectively non-terminating.

It therefore makes sense to eliminate general axioms from the KB whenever possible. Absorption is a technique that tries to do this by absorbing them into primitive definition axioms. The basic idea is that a general axiom of the form $C \sqsubseteq D$, where C may be a non-atomic concept, is manipulated (using the equivalences in Figure 9.4) so that it has the form of a primitive definition $\mathsf{A} \sqsubseteq D'$, where A is an atomic concept name. This axiom can then be merged into an existing primitive definition $\mathsf{A} \sqsubseteq C'$ to give $\mathsf{A} \sqsubseteq C' \sqcap D'$. For example, an axiom stating that all three

$$C_1 \sqcap C_2 \sqsubseteq D \iff C_1 \sqsubseteq D \sqcup \neg C_2$$
$$C \sqsubseteq D_1 \sqcap D_2 \iff C \sqsubseteq D_1 \text{ and } C \sqsubseteq D_2$$

Fig. 9.4. Axiom equivalences used in absorption.

sided geometric figures (i.e., triangles) also have three angles

$$\textsf{geometric-figure} \sqcap \exists \textsf{angles.three} \sqsubseteq \exists \textsf{sides.three}$$

could be transformed into an axiom stating that all geometric figures either have three sides or do not have three angles

$$\textsf{geometric-figure} \sqsubseteq \exists \textsf{sides.three} \sqcup \neg \exists \textsf{angles.three}$$

and then absorbed into the primitive definition of geometric figure ($\textsf{geometric-figure} \sqsubseteq \textsf{figure}$) to give

$$\textsf{geometric-figure} \sqsubseteq \textsf{figure} \sqcap (\exists \textsf{sides.three} \sqcup \neg \exists \textsf{angles.three}).$$

Given a KB divided into an unfoldable part $\mathcal{T}_u$ and a general part $\mathcal{T}_g$, the following procedure can be used to try to absorb the axioms from $\mathcal{T}_g$ into primitive definitions in $\mathcal{T}_u$. First a set $\mathcal{T}_g'$ is initialised to be empty, and any axioms $(C \equiv D) \in \mathcal{T}_g$ are replaced with an equivalent pair of axioms $C \sqsubseteq D$ and $\neg C \sqsubseteq \neg D$. Then for each axiom $(C \sqsubseteq D) \in \mathcal{T}_g$:

(i) Initialise a set $\mathbf{G} = \{\neg D, C\}$, representing the axiom in the form $\top \sqsubseteq \neg \sqcap \{\neg D, C\}$ (i.e. $\top \sqsubseteq D \sqcup \neg C$).

(ii) If for some $\textsf{A} \in \mathbf{G}$ there is a primitive definition axiom $(\textsf{A} \sqsubseteq C) \in \mathcal{T}_u$, then absorb the general axiom into the primitive definition axiom so that it becomes

$$\textsf{A} \sqsubseteq \sqcap \{C, \neg \sqcap (G \setminus \{\textsf{A}\})\},$$

and exit.

(iii) If for some $\textsf{A} \in \mathbf{G}$ there is an axiom $(\textsf{A} \equiv D) \in \mathcal{T}_u$, then substitute $\textsf{A} \in \mathbf{G}$ with D

$$\mathbf{G} \longrightarrow \{D\} \cup \mathbf{G} \setminus \{\textsf{A}\},$$

and return to step (ii).

(iv) If for some $\neg\textsf{A} \in \mathbf{G}$ there is an axiom $(\textsf{A} \equiv D) \in \mathcal{T}_u$, then substitute $\neg\textsf{A} \in \mathbf{G}$ with $\neg D$

$$\mathbf{G} \longrightarrow \{\neg D\} \cup \mathbf{G} \setminus \{\neg \textsf{A}\},$$

and return to step (ii).

(v) If there is some $C \in \mathbf{G}$ such that C is of the form $\sqcap\mathbf{S}$, then use associativity to simplify G

$$\mathbf{G} \longrightarrow \mathbf{S} \cup \mathbf{G} \setminus \{\sqcap\mathbf{S}\},$$

and return to step (ii).

(vi) If there is some $C \in \mathbf{G}$ such that C is of the form $\neg\sqcap\mathbf{S}$, then for every $D \in \mathbf{S}$ try to absorb (recursively)

$$\{\neg D\} \cup \mathbf{G} \setminus \{\neg\sqcap\mathbf{S}\},$$

and exit.

(vii) Otherwise, the axiom could not be absorbed, so add $\neg\sqcap\mathbf{G}$ to $\mathcal{T}_g'$

$$\mathcal{T}_g' \longrightarrow \mathcal{T}_g' \cup \neg\sqcap\mathbf{G},$$

and exit.

Note that this procedure allows parts of axioms to be absorbed. For example, given axioms $(\mathsf{A} \sqsubseteq D_1) \in \mathcal{T}_u$ and $(\mathsf{A} \sqcup \exists R.C \sqsubseteq D_2) \in \mathcal{T}_g$, then the general axiom would be partly absorbed into the definition axiom to give $(\mathsf{A} \sqsubseteq (D_1 \sqcap D_2)) \in \mathcal{T}_u$, leaving a smaller general axiom $(\neg\sqcap\{\neg D_2, \exists R.C\}) \in \mathcal{T}_g$.

When this procedure has been applied to all the axioms in $\mathcal{T}_g$, then $\mathcal{T}_g'$ represents those (parts of) axioms that could not be absorbed. The axioms in $\mathcal{T}_g'$ are already in the form $\top \sqsubseteq C$, so that $\sqcap\mathcal{T}_g'$ is the concept that must be added to every node in the tableaux expansion. This can be done using a universal role, as describe in Section 9.2.4, although in practice it may be simpler just to add the concept to the label of each newly created node.

The absorption process is clearly non-deterministic. In the first place, there may be more than one way to divide $\mathcal{T}$ into unfoldable and general parts. For example, if $\mathcal{T}$ contains multiple non-primitive definitions for some concept A, then one of them must be selected as a definition in $\mathcal{T}_u$ while the rest are treated as general axioms in $\mathcal{T}_g$. Moreover, the absorption procedure itself is non-deterministic as $\mathbf{G}$ may contain more than one primitive concept name into which the axiom could be absorbed. For example, in the case where $\{\mathsf{A}_1, \mathsf{A}_2\} = \mathbf{G}$, and there are two primitive definition axioms $\mathsf{A}_1 \sqsubseteq C$ and $\mathsf{A}_2 \sqsubseteq D$ in $\mathcal{T}_u$, then the axiom could be absorbed either into the definition of A_1 to give $\mathsf{A}_1 \sqsubseteq C \sqcap \neg\sqcap\{\mathsf{A}_2\}$ (equivalent to $\mathsf{A}_1 \sqsubseteq C \sqcap \neg\mathsf{A}_2$) or into the definition of A_2 to give $\mathsf{A}_2 \sqsubseteq C \sqcap \neg\sqcap\{\mathsf{A}_1\}$ (equivalent to $\mathsf{A}_2 \sqsubseteq C \sqcap \neg\mathsf{A}_1$).

It would obviously be sensible to choose the "best" absorption (the one that maximised empirical tractability), but it is not clear how to do this—in fact it is not even clear how to define "best" in this context [Horrocks and Tobies, 2000]. If $\mathcal{T}$ contains more than one definition axiom for a given concept name, then empirical evidence suggests that efficiency is improved by retaining as many non-primitive definition

axioms in $\mathcal{T}_u$ as possible. Another intuitively obvious possibility is to preferentially absorb into the definition axiom of the most specific primitive concept, although this only helps in the case that $\mathsf{A}_1 \sqsubseteq \mathsf{A}_2$ or $\mathsf{A}_2 \sqsubseteq \mathsf{A}_1$. Other more sophisticated schemes might be possible, but have yet to be investigated.

The advantages of absorption are:

- It can lead to a dramatic improvement in performance. For example, without absorption, satisfiability of the Galen KB (i.e., the satisfiability of $\top$) could not be proved by either FaCT or Dlp, even after several weeks of CPU time. After absorption, the problem becomes so trivial that the CPU time required is hard to measure.
- It is logic and algorithm independent.

The disadvantage is the overhead required for the pre-processing, although this is generally small compared to classification times. However, the procedure described is almost certainly sub-optimal, and trying to find an optimal absorption may be much more costly.

9.5.2 Optimising classification

DL systems are often used to classify a KB, that is to compute a partial ordering or *hierarchy* of named concepts in the KB based on the subsumption relationship. As subsumption testing is always potentially costly, it is important to ensure that the classification process uses the smallest possible number of tests. Minimising the number of subsumption tests required to classify a concept in the concept hierarchy can be treated as an abstract order-theoretic problem which is independent of the ordering relation. However, some additional optimisation can be achieved by using the structure of concepts to reveal obvious subsumption relationships and to control the order in which concepts are added to the hierarchy (where this is possible).

The concept hierarchy is usually represented by a directed acyclic graph where nodes are labelled with sets of concept names (because multiple concept names may be logically equivalent), and edges correspond with subsumption relationships. The subsumption relation is both transitive and reflexive, so a classified concept A subsumes a classified concept B if either:

(i) both A and B are in the label of some node x, or
(ii) A is in the label of some node x, there is an edge $\langle x, y\rangle$ in the graph, and the concept(s) in the label of node y subsume B.

It will be assumed that the hierarchy always contains a top node (a node whose label includes $\top$) and a bottom node (a node whose label includes $\bot$) such that the

top node subsumes the bottom node. If the KB is unsatisfiable then the hierarchy will consist of a single node whose label includes both $\top$ and $\bot$.

Algorithms based on traversal of the concept hierarchy can be used to minimise the number of tests required in order to add a new concept [Baader *et al.*, 1992a]. The idea is to compute a concept's subsumers by searching down the hierarchy from the top node (the *top search* phase) and its subsumees by searching up the hierarchy from the bottom node (the *bottom search* phase).

When classifying a concept A, the top search takes advantage of the transitivity of the subsumption relation by propagating failed results down the hierarchy. It concludes, without performing a subsumption test, that if A is not subsumed by B, then it cannot be subsumed by any other concept that is subsumed by B:

$$\mathcal{T} \not\models \mathsf{A} \sqsubseteq \mathsf{B} \text{ and } \mathcal{T} \models \mathsf{B}' \sqsubseteq \mathsf{B} \quad \text{implies} \quad \mathcal{T} \not\models \mathsf{A} \sqsubseteq \mathsf{B}'$$

To maximise the effect of this strategy, a modified breadth first search is used [Ellis, 1992] which ensures that a test to discover if B subsumes A is never performed until it has been established that A is subsumed by all of the concepts known to subsume B.

The bottom search uses a corresponding technique, testing if A subsumes B only when A is already known to subsume all those concepts that are subsumed by B. Information from the top search is also used by confining the bottom search to those concepts which are subsumed by all of A's subsumers.

This abstract partial ordering technique can be enhanced by taking advantage of the structure of concepts and the axioms in the KB. If the KB contains an axiom $\mathsf{A} \sqsubseteq C$ or $\mathsf{A} \equiv C$, then C is said to be a *told subsumer* of A. If C is a conjunctive concept $(C_1 \sqcap \ldots \sqcap C_n)$, then from the structural subsumption relationship

$$D \sqsubseteq (C_1 \sqcap \cdots \sqcap C_n) \quad \text{implies} \quad D \sqsubseteq C_1 \text{ and } \cdots \text{ and } D \sqsubseteq C_n$$

it is possible to conclude that $C_1, \ldots, C_n$ are also told subsumers of A. Moreover, due to the transitivity of the subsumption relation, any told subsumers of $C_1, \ldots, C_n$ are also told subsumers of A. Before classifying A, all of its told subsumers which have already been classified, and all their subsumers, can be marked as subsumers of A; subsumption tests with respect to these concepts are therefore rendered unnecessary. This idea can be extended in the obvious way to take advantage of a structural subsumption relationship with respect to disjunctive concepts,

$$(C_1 \sqcup \cdots \sqcup C_n) \sqsubseteq D \quad \text{implies} \quad C_1 \sqsubseteq D \text{ and } \cdots \text{ and } C_n \sqsubseteq D.$$

If the KB contains an axiom $\mathsf{A} \equiv C$ and C is a disjunctive concept $(C_1 \sqcup \ldots \sqcup C_n)$, then A is a told subsumer of $C_1, \ldots, C_n$.

To maximise the effect of the told subsumer optimisation, concepts should be classified in *definition order*. This means that a concept A is not classified until all

of its told subsumers have been classified. When classifying an unfoldable KB, this ordering can be exploited by omitting the bottom search phase for primitive concept names and assuming that they only subsume (concepts equivalent to) $\bot$. This is possible because, with an unfoldable KB, a primitive concept can only subsume concepts for which it is a told subsumer. Therefore, as concepts are classified in definition order, a primitive concept will always be classified before any of the concepts that it subsumes. This additional optimisation cannot be used with a general KB because, in the presence of general axioms, it can no longer be guaranteed that a primitive concept will only subsume concepts for which it is a told subsumer. For example, given a KB $\mathcal{T}$ such that

$$\mathcal{T} = \{\mathsf{A} \sqsubseteq \exists R.C,\ \exists R.C \sqsubseteq \mathsf{B}\},$$

then B is not a told subsumer of A, and A may be classified first. However, when B is classified the bottom search phase will discover that it subsumes A due to the axiom $\exists R.C \sqsubseteq \mathsf{B}$.

The advantages of the enhanced traversal classification method are:

- It can significantly reduce the number of subsumption tests required in order to classify a KB [Baader *et al.*, 1992a].
- It is logic and (subsumption) algorithm independent.

There appear to be few disadvantages to this method, and it is used (in some form) in most implemented DL systems.

9.5.3 Optimising subsumption testing

The classification optimisations described in Section 9.5.2 help to reduce the number of subsumption tests that are performed when classifying a KB, and the combination of normalisation, simplification and lazy unfolding facilitates the detection of "obvious" subsumption relationships by allowing unsatisfiability to be rapidly demonstrated. However, detecting "obvious" non-subsumption (satisfiability) is more difficult for tableaux algorithms. This is unfortunate as concept hierarchies from realistic applications are typically broad, shallow and tree-like. The top search phase of classifying a new concept A in such a hierarchy will therefore result in several subsumption tests being performed at each node, most of which are likely to fail. These failed tests could be very costly (if, for example, proving the satisfiability of A is a hard problem), and they could also be very repetitive.

This problem can be tackled by trying to use cached results from previous tableaux tests to prove non-subsumption without performing a new satisfiability

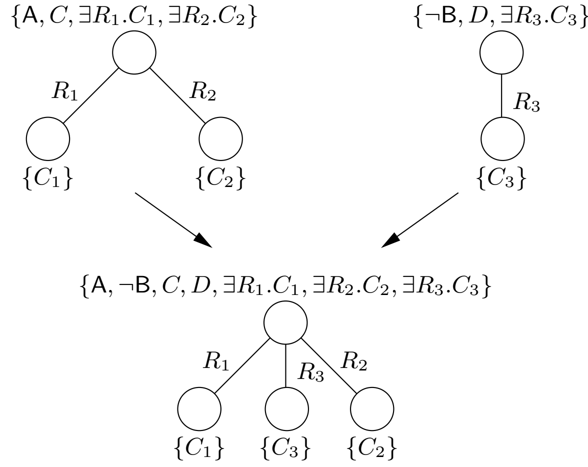

Fig. 9.5. Joining expansion trees for A and ¬B.

test. For example, given two concepts A and B defined by the axioms

$$\begin{aligned} \mathsf{A} &\equiv C \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2, \text{ and} \\ \mathsf{B} &\equiv \neg D \sqcup \forall R_3.\neg C_3, \end{aligned}$$

then A is not subsumed by B if the concept $\mathsf{A} \sqcap \neg \mathsf{B}$ is satisfiable. If tableaux expansion trees for A and ¬B have already been cached, then the satisfiability of the conjunction can be demonstrated by a tree consisting of the trees for A and ¬B joined at their root nodes, as shown in Figure 9.5 (note that $\neg \mathsf{B} \equiv D \sqcap \exists R_3.C_3$).

Given two fully expanded and clash free tableaux expansion trees $\mathbf{T}_1$ and $\mathbf{T}_2$ representing models of (satisfiable) concepts A and ¬B respectively, the tree created by joining $\mathbf{T}_1$ and $\mathbf{T}_2$ at their root nodes is a fully expanded and clash free tree representing a model of $\mathsf{A} \sqcap \neg \mathsf{B}$ provided that the union of the root node labels does not contain a clash and that no tableaux expansion rules are applicable to the new tree. For most logics, this can be ascertained by examining the labels of the root nodes and the labels of the edges connection them with their successors. With the $\mathcal{ALC}$ logic for example, if x_1 and x_2 are the two root nodes, then the new tree will be fully expanded and clash free provided that

(i) the union of the root node labels does not contain an immediate contradiction, i.e., there is no C such that $\{C, \neg C\} \subseteq \mathcal{L}(x_1) \cup \mathcal{L}(x_2)$, and

(ii) there is no interaction between value restrictions in the label of one root node and edges connecting the other root node with its successors that might make the $\forall$-rule applicable to the joined tree, i.e., there is no R such that $\forall R.C \in \mathcal{L}(x_1)$ and $\mathbf{T}_2$ has an edge $\langle x_2, y \rangle$ with $\mathcal{L}(\langle x_2, y \rangle) = R$, or $\forall R.C \in \mathcal{L}(x_2)$ and $\mathbf{T}_1$ has an edge $\langle x_1, y \rangle$ with $\mathcal{L}(\langle x_1, y \rangle) = R$.

With more expressive logics it may be necessary to consider other interactions that could lead to the application of tableaux expansion rules. With a logic that included number restrictions, for example, it would be necessary to check that these could not be violated by the root node successors in the joined tree.

It would be possible to join trees in a wider range of cases by examining the potential interactions in more detail. For example, a value restriction $\forall R.C \in \mathcal{L}(x_1)$ and an R labelled edge $\langle x_2, y\rangle$ would not make the $\forall$-rule applicable to the joined tree if $C \in \mathcal{L}(x_2)$. However, considering only root nodes and edges provides a relatively fast test and reduces the storage required by the cache. Both the time required by the test and the size of the cache can be reduced even further by only storing relevant components of the root node labels and edges from the fully expanded and clash free tree that demonstrates the satisfiability of a concept. In the case of $\mathcal{ALC}$, the relevant components from a tree demonstrating the satisfiability of a concept A are the set of (possibly negated) atomic concept names in the root node label (denoted $\mathcal{L}_c(\mathsf{A})$), the set of role names in value restrictions in the root node label (denoted $\mathcal{L}_\forall(\mathsf{A})$), and the set of role names labelling edges connecting the root node with its successors (denoted $\mathcal{L}_\exists(\mathsf{A})$).[1] These components can be cached as a triple $(\mathcal{L}_c(\mathsf{A}), \mathcal{L}_\forall(\mathsf{A}), \mathcal{L}_\exists(\mathsf{A}))$.

When testing if A is subsumed by B, the algorithm can now proceed as follows.

(i) If any of $(\mathcal{L}_c(\mathsf{A}), \mathcal{L}_\forall(\mathsf{A}), \mathcal{L}_\exists(\mathsf{A}))$, $(\mathcal{L}_c(\neg\mathsf{A}), \mathcal{L}_\forall(\neg\mathsf{A}), \mathcal{L}_\exists(\neg\mathsf{A}))$, $(\mathcal{L}_c(\mathsf{B}), \mathcal{L}_\forall(\mathsf{B}), \mathcal{L}_\exists(\mathsf{B}))$ or $(\mathcal{L}_c(\neg\mathsf{B}), \mathcal{L}_\forall(\neg\mathsf{B}), \mathcal{L}_\exists(\neg\mathsf{B}))$ are not in the cache, then perform the appropriate satisfiability tests and update the cache accordingly. In the case where a concept C is unsatisfiable, $\mathcal{L}_c(C) = \{\bot\}$ and $\mathcal{L}_c(\neg C) = \{\top\}$.

(ii) Conclude that $\mathsf{A} \sqsubseteq \mathsf{B}$ ($\mathsf{A} \sqcap \neg\mathsf{B}$ is not satisfiable) if $\mathcal{L}_c(\mathsf{A}) = \{\bot\}$ or $\mathcal{L}_c(\mathsf{B}) = \{\top\}$.

(iii) Conclude that $\mathsf{A} \not\sqsubseteq \mathsf{B}$ ($\mathsf{A} \sqcap \neg\mathsf{B}$ is satisfiable) if

  (a) $\mathcal{L}_c(\mathsf{A}) = \{\top\}$ and $\mathcal{L}_c(\mathsf{B}) \neq \{\top\}$, or

  (b) $\mathcal{L}_c(\mathsf{A}) \neq \{\bot\}$ and $\mathcal{L}_c(\mathsf{B}) = \{\bot\}$, or

  (c) $\mathcal{L}_\forall(\mathsf{A}) \sqcap \mathcal{L}_\exists(\mathsf{B}) = \emptyset$, $\mathcal{L}_\forall(\mathsf{B}) \sqcap \mathcal{L}_\exists(\mathsf{A}) = \emptyset$, $\bot \notin \mathcal{L}_c(\mathsf{A}) \cup \mathcal{L}_c(\mathsf{B})$, and there is no C such that $\{C, \neg C\} \subseteq \mathcal{L}_c(\mathsf{A}) \cup \mathcal{L}_c(\mathsf{B})$.

(iv) Otherwise perform a satisfiability test on $\mathsf{A} \sqcap \neg\mathsf{B}$, concluding that $\mathsf{A} \sqsubseteq \mathsf{B}$ if it is not satisfiable and that $\mathsf{A} \not\sqsubseteq \mathsf{B}$ if it is satisfiable.

When a concept A is added to the hierarchy, this procedure will result in satisfiability tests immediately being performed for both A and $\neg\mathsf{A}$. During the subsequent top search phase, at each node x in the hierarchy such that some $C \in \mathcal{L}(x)$ subsumes A, it will be necessary to perform a subsumption test for each subsumee

[1] Consideration can be limited to atomic concept names because expanded conjunction and disjunction concepts are no longer relevant to the validity of the tree, and are only retained in order to facilitate early clash detection.

node y (unless some of them can be avoided by the classification optimisations discussed in Section 9.5.2). Typically only one of these subsumption tests will lead to a full satisfiability test being performed, the rest being shown to be obvious non-subsumptions using the cached partial trees. Moreover, the satisfiability test that is performed will often be an "obvious" subsumption, and unsatisfiability will rapidly be demonstrated.

The optimisation is less useful during the bottom search phase as nodes in the concept hierarchy are typically connected to only one subsuming node. The exception to this is the bottom ($\bot$) node, which may be connected to a very large number of subsuming nodes. Again, most of the subsumption tests that would be required by these nodes can be avoided by demonstrating non-subsumption using cached partial trees.

The caching technique can be extended in order to avoid the construction of obviously satisfiable and unsatisfiable sub-trees during tableaux expansion. For example, if some leaf node x is about to be expanded, and $\mathcal{L}(x) = \{\mathsf{A}\}$, unfolding and expanding $\mathcal{L}(x)$ is clearly unnecessary if A is already known to be either satisfiable (i.e., $(\mathcal{L}_c(\mathsf{A}), \mathcal{L}_\forall(\mathsf{A}), \mathcal{L}_\exists(\mathsf{A}))$ is in the cache and $\mathcal{L}_c(\mathsf{A}) \neq \{\bot\}$) or unsatisfiable (i.e., $(\mathcal{L}_c(\mathsf{A}), \mathcal{L}_\forall(\mathsf{A}), \mathcal{L}_\exists(\mathsf{A}))$ is in the cache and $\mathcal{L}_c(\mathsf{A}) = \{\bot\}$).

This idea can be further extended by caching (when required) partial trees for all the syntactically distinct concepts discovered by the normalisation and simplification process, and trying to join cached partial trees for all the concepts in a leaf node's label before starting the expansion process. For example, with the logic $\mathcal{ALC}$ and a node x such that

$$\mathcal{L}(x) = \{C_1, \ldots, C_n\},$$

x is unsatisfiable if for some $1 \leqslant i \leqslant n$, $\mathcal{L}_c(C_i) = \{\bot\}$, and x is satisfiable if for all $1 \leqslant i \leqslant n$ and $1 < j \leqslant n$,

(i) $\mathcal{L}_\forall(C_i) \sqcap \mathcal{L}_\exists(C_j) = \emptyset$,
(ii) $\mathcal{L}_\exists(C_i) \sqcap \mathcal{L}_\forall(C_j) = \emptyset$, and
(iii) there is no C such that $\{C, \neg C\} \subseteq \mathcal{L}_c(C_i) \cup \mathcal{L}_c(C_j)$.

As before, additional interactions may need to be considered with more expressive logics. Moreover, with logics that support inverse roles, the effect that the sub-tree might have on its predecessor must also be considered. For example, if x is an R-successor of some node y, and $R^- \in \mathcal{L}_\forall(C_i)$ for one of the $C_i \in \mathcal{L}(x)$, then the expanded $\mathcal{L}(x)$ represented by the cached partial trees would contain a value restriction of the form $\forall R^-.D$ that could augment $\mathcal{L}(y)$.

The advantages of caching partial tableaux expansion trees are:

- When classifying a realistic KB, most satisfiability tests can be avoided. For

example, the number of satisfiability tests performed by the FaCT system when classify the Galen KB is reduced from 122,695 to 23,492, a factor of over 80%.
- Without caching, some of the most costly satisfiability tests are repeated (with minor variations) many times. The time saving due to caching is therefore even greater than the saving in satisfiability tests.

The disadvantages are:

- The overhead of performing satisfiability tests on individual concepts and their negations in order to generate the partial trees that are cached.
- The overhead of storing the partial trees. This is not too serious a problem as the number of trees cached is equal to the number of named concepts in the KB (or the number of syntactically distinct concepts if caching is used in sub-problems).
- The overhead of determining if the cached partial trees can be merged, which is wasted if they cannot be.
- Its main use is when classifying a KB, or otherwise performing many similar satisfiability tests. It is of limited value when performing single tests.

9.5.4 Optimising satisfiability testing

In spite of the various techniques outlined in the preceding sections, at some point the DL system will be forced to perform a "real" subsumption test, which for a tableaux based system means testing the satisfiability of a concept. For expressive logics, such tests can be very costly. However, a range of optimisations can be applied that dramatically improve performance in typical cases. Most of these are aimed at reducing the size of the search space explored by the algorithm as a result of applying non-deterministic tableaux expansion rules.

9.5.4.1 Semantic branching search

Standard tableaux algorithms use a search technique based on *syntactic branching*. When expanding the label of a node x, syntactic branching works by choosing an unexpanded disjunction $(C_1 \sqcup \ldots \sqcup C_n)$ in $\mathcal{L}(x)$ and searching the different models obtained by adding each of the disjuncts $C_1, \ldots, C_n$ to $\mathcal{L}(x)$ [Giunchiglia and Sebastiani, 1996b]. As the alternative branches of the search tree are not disjoint, there is nothing to prevent the recurrence of an unsatisfiable disjunct in different branches. The resulting wasted expansion could be costly if discovering the unsatisfiability requires the solution of a complex sub-problem. For example, tableaux expansion of a node x, where $\{(A \sqcup B), (A \sqcup C)\} \subseteq \mathcal{L}(x)$ and A is an unsatisfiable concept, could lead to the search pattern shown in Figure 9.6, in which the unsatisfiability of $\mathcal{L}(x) \cup A$ must be demonstrated twice.

This problem can be dealt with by using a *semantic branching* technique adapted

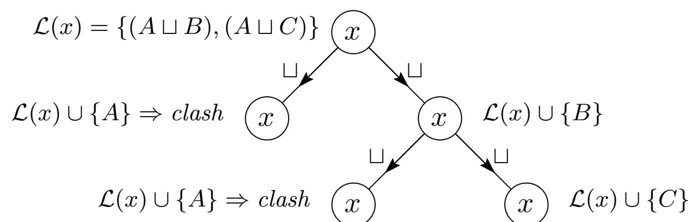

Fig. 9.6. Syntactic branching search.

from the Davis-Putnam-Logemann-Loveland procedure (DPLL) commonly used to solve propositional satisfiability (SAT) problems [Davis and Putnam, 1960; Davis *et al.*, 1962; Freeman, 1996].[1] Instead of choosing an unexpanded disjunction in $\mathcal{L}(x)$, a single disjunct D is chosen from one of the unexpanded disjunctions in $\mathcal{L}(x)$. The two possible sub-trees obtained by adding either D or $\neg D$ to $\mathcal{L}(x)$ are then searched. Because the two sub-trees are strictly disjoint, there is no possibility of wasted search as in syntactic branching. Note that the order in which the two branches are explored is irrelevant from a theoretical viewpoint, but may offer further optimisation possibilities (see Section 9.5.4.4).

The advantages of semantic branching search are:

- A great deal is known about the implementation and optimisation of the DPLL algorithm. In particular, both *local simplification* (see Section 9.5.4.2) and *heuristic guided search* (see Section 9.5.4.4) can be used to try to minimise the size of the search tree (although it should be noted that both these techniques can also be adapted for use with syntactic branching search).
- It can be highly effective with some problems, particularly randomly generated problems [Horrocks and Patel-Schneider, 1999].

The disadvantages are:

- It is possible that performance could be degraded by adding the negated disjunct in the second branch of the search tree, for example if the disjunct is a very large or complex concept. However this does not seem to be a serious problem in practice, with semantic branching rarely exhibiting significantly worse performance than syntactic branching.
- Its effectiveness is problem dependent. It is most effective with randomly generated problems, particularly those that are over-constrained (likely to be unsatisfiable). It is also effective with some of the hand crafted problems from the Tableaux'98 benchmark suite. However, it appears to be of little benefit when classifying realistic KBs [Horrocks and Patel-Schneider, 1998a].

[1] An alternative solution is to enhance syntactic branching with "no-good" lists in order to avoid reselecting a known unsatisfiable disjunct [Donini and Massacci, 2000].

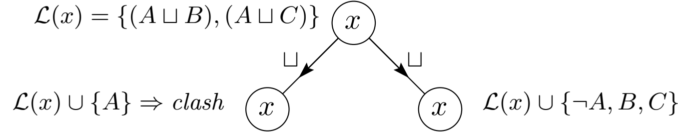

Fig. 9.7. Semantic branching search.

9.5.4.2 Local simplification

Local simplification is another technique used to reduce the size of the search space resulting from the application of non-deterministic expansion rules. Before any non-deterministic expansion of a node label $\mathcal{L}(x)$ is performed, disjunctions in $\mathcal{L}(x)$ are examined, and if possible simplified. The simplification most commonly used (although by no means the only one possible) is to deterministically expand disjunctions in $\mathcal{L}(x)$ that present only one expansion possibility and to detect a clash when a disjunction in $\mathcal{L}(x)$ has no expansion possibilities. This simplification has been called Boolean constraint propagation (BCP) [Freeman, 1995]. In effect, the inference rules

$$\frac{\neg C_1,\ldots,\neg C_n, C_1 \sqcup \ldots \sqcup C_n \sqcup D}{D} \quad \text{and} \quad \frac{C_1,\ldots,C_n, \neg C_1 \sqcup \ldots \sqcup \neg C_n \sqcup D}{D}$$

are being used to simplify the conjunctive concept represented by $\mathcal{L}(x)$.

For example, given a node x such that

$$\{(C \sqcup (D_1 \sqcap D_2)), (\neg D_1 \sqcup \neg D_2 \sqcup C), \neg C\} \subseteq \mathcal{L}(x),$$

BCP deterministically expands the disjunction $(C \sqcup (D_1 \sqcap D_2))$, adding $(D_1 \sqcap D_2)$ to $\mathcal{L}(x)$, because $\neg C \in \mathcal{L}(x)$. The deterministic expansion of $(D_1 \sqcap D_2)$ adds both D_1 and D_2 to $\mathcal{L}(x)$, allowing BCP to identify $(\neg D_1 \sqcup \neg D_2 \sqcup C)$ as a clash (without any branching having occurred), because $\{D_1, D_2, \neg C\} \subseteq \mathcal{L}(x)$.

BCP simplification is usually described as an integral part of SAT based algorithms [Giunchiglia and Sebastiani, 1996a], but it can also be used with syntactic branching. However, it is more effective with semantic branching as the negated concepts introduced by failed branches can result in additional simplifications. Taking the above example of $\{(A \sqcup B), (A \sqcup C)\} \subseteq \mathcal{L}(x)$, adding $\neg A$ to $\mathcal{L}(x)$ allows BCP to deterministically expand both of the disjunctions using the simplifications $(A \sqcup B)$ and $\neg A$ implies B and $(A \sqcup C)$ and $\neg A$ implies C. The reduced search space resulting from the combination of semantic branching and BCP is shown in Figure 9.7.

The advantages of local simplification are:

- It is applicable to a wide range of logics and algorithms.
- It can never increase the size of the search space.

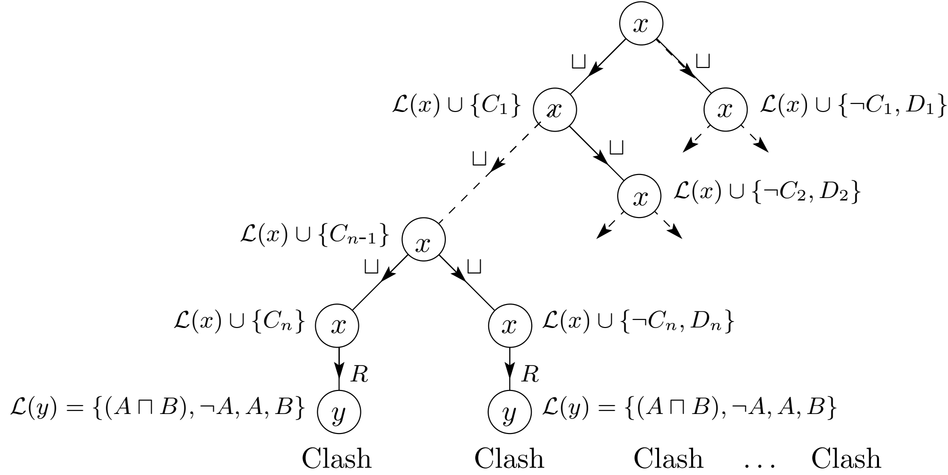

Fig. 9.8. Thrashing in backtracking search.

The disadvantages are:

- It may be costly to perform without using complex data structures [Freeman, 1995].
- Its effectiveness is relatively limited and problem dependant. It is most effective with randomly generated problems, particularly those that are over-constrained [Horrocks and Patel-Schneider, 1998a].

9.5.4.3 Dependency directed backtracking

Inherent unsatisfiability concealed in sub-problems can lead to large amounts of unproductive backtracking search, sometimes called thrashing. The problem is exacerbated when blocking is used to guarantee termination, because blocking may require that sub-problems only be explored after all other forms of expansion have been performed. For example, expanding a node x (using semantic branching), where

$$\mathcal{L}(x) = \{(C_1 \sqcup D_1), \ldots, (C_n \sqcup D_n), \exists R.(A \sqcap B), \forall R.\neg A\},$$

could lead to the fruitless exploration of 2^n possible R-successors of x before the inherent unsatisfiability is discovered (note that if $\mathcal{L}(x)$ simply included $\exists R.A$ instead of $\exists R.(A \sqcap B)$, then the inherent unsatisfiability would have been detected immediately due to the normalisation of $\exists R.A$ as $\neg\forall R.\neg A$). The search tree resulting from the tableaux expansion is illustrated in Figure 9.8.

This problem can be addressed by identifying the causes of clashes, and using this information to prune or restructure the search space—a technique known as *dependency directed backtracking*. The form most commonly used in practice, called *backjumping*, is adapted from a technique that has been used in solving constraint

satisfiability problems [Baker, 1995] (a similar technique was also used in the Harp theorem prover [Oppacher and Suen, 1988]). Backjumping works by labelling each concept in a node label and each role in an edge label with a dependency set indicating the branching points on which it depends. A concept $C \in \mathcal{L}(x)$ depends on a branching point if C was added to $\mathcal{L}(x)$ at the branching point or if C depends on another concept D (or role R), and D (or R) depends on the branching point. A concept $C \in \mathcal{L}(x)$ depends on a concept D (or role R) when C was added to $\mathcal{L}(x)$ by the application of a deterministic expansion rule that used D (or R); a role $R = \mathcal{L}(\langle x, y \rangle)$ depends on a concept D when $\langle x, y \rangle$ was labelled R by the application of a deterministic expansion rule that used D. For example, if $A \in \mathcal{L}(y)$ was derived from the expansion of $\forall R.A \in \mathcal{L}(x)$, then $A \in \mathcal{L}(y)$ depends on both $\forall R.A \in \mathcal{L}(x)$ and $R = \mathcal{L}(\langle x, y \rangle)$.

Labelling roles with dependency sets can be avoided in algorithms where a combined $\exists\forall$-rule is used, as the dependency sets for concepts in the label of the new node can be derived in a single step. On the other hand, more complex algorithms and optimisation techniques may lead to more complex dependencies. For example, if $C_n \in \mathcal{L}(x)$ was derived from a BCP simplification of $\{(C_1 \sqcup \ldots \sqcup C_n), \neg C_1, \ldots, \neg C_{n-1}\} \subseteq \mathcal{L}(x)$, then it depends on the disjunction $(C_1 \sqcup \ldots \sqcup C_n)$ and all of $\neg C_1, \ldots, \neg C_{n-1}$.

When a clash is discovered, the dependency sets of the clashing concepts can be used to identify the most recent branching point where exploring the other branch might alleviate the cause of the clash. It is then possible to jump back over intervening branching points *without* exploring any alternative branches. Again, more complex algorithms and optimisations may lead to more complex dependencies. For example, if the clash results from a BCP simplification of $\{(C_1 \sqcup \ldots \sqcup C_n), \neg C_1, \ldots, \neg C_n\} \subseteq \mathcal{L}(x)$, then it depends on the disjunction $(C_1 \sqcup \ldots \sqcup C_n)$ and all of $\neg C_1, \ldots, \neg C_n$.

When testing the satisfiability of a concept C, the dependency set of $C \in \mathcal{L}(x)$ is initialised to $\emptyset$ (the empty set) and a branching depth counter b is initialised to 1. The search algorithm then proceeds as follows:

(i) Perform deterministic expansion, setting the dependency set of each concept added to a node label and each role assigned to an edge label to the union of the dependency sets of the concepts and roles on which they depend.

   (a) If a clash is discovered, then return the union of the dependency sets of the clashing concepts.

   (b) If a clash free expansion is discovered, then return $\{0\}$.

(ii) Branch on a concept $D \in \mathcal{L}(y)$, trying first $\mathcal{L}(y) \cup \{D\}$ and then $\mathcal{L}(y) \cup \{\neg D\}$.

   (a) Add D to $\mathcal{L}(y)$ with a dependency set $\{b\}$, and increment b.

(b) Set $\mathbf{D}_1$ to the dependency set returned by a recursive call to the search algorithm, and decrement b.

(c) If $b \notin \mathbf{D}_1$, then return $\mathbf{D}_1$ *without* exploring the second branch.

(d) If $b \in \mathbf{D}_1$, then add $\neg D$ to $\mathcal{L}(y)$ with a dependency set $\mathbf{D}_1 \setminus \{b\}$ and return to step (i).

If the search returns $\{0\}$, then a successful expansion was discovered and the algorithm returns "satisfiable", otherwise all possible expansions led to a clash and "unsatisfiable" is returned.

Let us consider the earlier example and suppose that $\exists R.(A \sqcap B)$ has a dependency set $\mathbf{D}_i$, $\forall R.\neg A$ has a dependency set $\mathbf{D}_j$ and $b = k$ (meaning that there had already been $k-1$ branching points in the search tree). Note that the largest values in $\mathbf{D}_i$ and $\mathbf{D}_j$ must be less than k, as neither concept can depend on a branching point that has not yet been reached.

At the kth branching point, C_1 is added to $\mathcal{L}(x)$ with a dependency set $\{k\}$ and b is incremented. The search continues in the same way until the $(k+n-1)$th branching point, when C_n is added to $\mathcal{L}(x)$ with a dependency set $\{k+n-1\}$. Next, $\exists R.(A \sqcap B)$ is deterministically expanded, generating an R-successor y with $R = \langle x, y\rangle$ labelled $\mathbf{D}_i$ and $(A \sqcap B) \in \mathcal{L}(y)$ labelled $\mathbf{D}_i$. Finally, $\forall R.\neg A$ is deterministically expanded, adding $\neg A$ to $\mathcal{L}(y)$ with a label $\mathbf{D}_i \cup \mathbf{D}_j$ (because it depends on both $\forall R.\neg A \in \mathcal{L}(x)$ and $R = \langle x, y\rangle$).

The expansion now continues with $\mathcal{L}(y)$, and $(A \sqcap B)$ is deterministically expanded, adding A and B to $\mathcal{L}(y)$, both labelled $\mathbf{D}_i$. This results in a clash as $\{A, \neg A\} \subseteq \mathcal{L}(y)$, and the set $\mathbf{D}_i \cup \mathbf{D}_i \cup \mathbf{D}_j = \mathbf{D}_i \cup \mathbf{D}_j$ (the union of the dependency sets from the two clashing concepts) is returned. The algorithm will then backtrack through each of the preceding n branching points without exploring the second branches, because in each case $b \notin \mathbf{D}_i \cup \mathbf{D}_j$ (remember that the largest values in $\mathbf{D}_i$ and $\mathbf{D}_j$ are less than k), and will continue to backtrack until it reaches the branching point equal to the maximum value in $\mathbf{D}_i \cup \mathbf{D}_j$ (if $\mathbf{D}_i = \mathbf{D}_j = \emptyset$, then the algorithm will backtrack through all branching points and return "unsatisfiable"). Figure 9.9 illustrates the pruned search tree, with the number of R-successors explored being reduced by $2^n - 1$.

Backjumping can also be used with syntactic branching, but the procedure is slightly more complex as there may be more than two possible choices at a given branching point, and the dependency set of the disjunction being expanded must also be taken into account. When expanding a disjunction of size n with a dependency set $\mathbf{D}_d$, the first $n-1$ disjuncts are treated like the first branch in the semantic branching algorithm, an immediate backtrack occurring if the recursive search discovers a clash that does not depend on b. If each of these branches re-

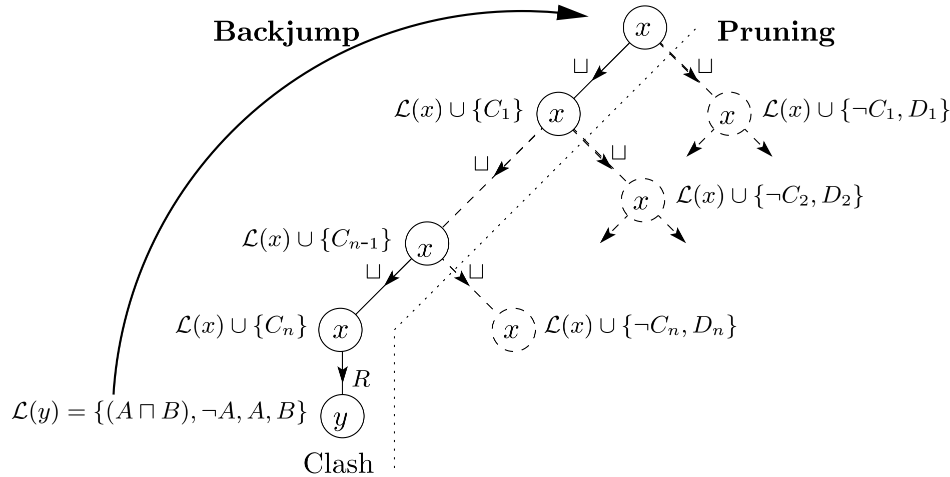

Fig. 9.9. Pruning the search using backjumping.

turns a dependency set $\mathbf{D}_i$ such that $b \in \mathbf{D}_i$, then the nth disjunct is added with a dependency set $(\mathbf{D}_1 \cup \ldots \cup \mathbf{D}_{n-1} \cup \mathbf{D}_d) \setminus b$.

The advantages of backjumping are

- It can lead to a dramatic reduction in the size of the search tree and thus a huge performance improvement. For example, when trying to classify the Galen model using either FaCT or Dlp with backjumping disabled, single satisfiability tests were encountered that could not be solved even after several weeks of CPU time.
- The size of the search space can never be increased.

The disadvantage is the overhead of propagating and storing the dependency sets. The storage overhead can be alleviated to some extent by using a pointer based implementation so that propagating a dependency set only requires the copying of a pointer. A simpler scheme using single maximal dependency values instead of sets would also be possible, but some dependency information would be lost and this could lead to less efficient pruning of the search tree.

9.5.4.4 Heuristic guided search

Heuristic techniques can be used to guide the search in a way that tries to minimise the size of the search tree. A method that is widely used in DPLL SAT algorithms is to branch on the disjunct that has the Maximum number of Occurrences in disjunctions of Minimum Size—the well known MOMS heuristic [Freeman, 1995]. By choosing a disjunct that occurs frequently in small disjunctions, the MOMS heuristic tries to maximise the effect of BCP. For example, if the label of a node x contains the unexpanded disjunctions $C \sqcup D_1, \ldots, C \sqcup D_n$, then branching on C

leads to their deterministic expansion in a single step: when C is added to $\mathcal{L}(x)$, all of the disjunctions are fully expanded and when $\neg C$ is added to $\mathcal{L}(x)$, BCP will expand all of the disjunctions, causing $D_1, \ldots, D_n$ to be added to $\mathcal{L}(x)$. Branching first on any of $D_1, \ldots, D_n$, on the other hand, would only cause a single disjunction to be expanded.

The MOMS value for a candidate concept C is computed simply by counting the number of times C or its negation occur in minimally sized disjunctions. There are several variants of this heuristic, including the heuristic from Jeroslow and Wang [Jeroslow and Wang, 1990]. The Jeroslow and Wang heuristic considers all occurrences of a disjunct, weighting them according to the size of the disjunction in which they occur. The heuristic then selects the disjunct with the highest overall weighting, again with the objective of maximising the effect of BCP and reducing the size of the search tree.

When a disjunct C has been selected from the disjunctions in $\mathcal{L}(x)$, a BCP maximising heuristic can also be used to determine the order in which the two possible branches, $\mathcal{L}(x) \cup \{C\}$ and $\mathcal{L}(x) \cup \{\neg C\}$, are explored. This is done by separating the two components of the heuristic weighting contributed by occurrences of C and $\neg C$, trying $\mathcal{L}(x) \cup \{C\}$ first if C made the *smallest* contribution, and trying $\mathcal{L}(x) \cup \{\neg C\}$ first otherwise. The intention is to prune the search tree by maximising BCP in the first branch.

Unfortunately, MOMS-style heuristics can interact adversely with the backjumping optimisation because they do not take dependency information into account. This was first discovered in the FaCT system, when it was noticed that using MOMS heuristic often led to much worse performance. The cause of this phenomenon turned out to be the fact that, without the heuristic, the data structures used in the implementation naturally led to "older" disjunctions (those dependent on earlier branching points) being expanded before "newer" ones, and this led to more effective pruning if a clash was discovered. Using the heuristic disturbed this ordering and reduced the effectiveness of backjumping [Horrocks, 1997b].

Moreover, MOMS-style heuristics are of little value themselves in description logic systems because they rely for their effectiveness on finding the same disjuncts recurring in multiple unexpanded disjunctions: this is likely in hard propositional problems, where the disjuncts are propositional variables, and where the number of different variables is usually small compared to the number of disjunctive clauses (otherwise problems would, in general, be trivially satisfiable); it is unlikely in concept satisfiability problems, where the disjuncts are (possibly non-atomic) concepts, and where the number of different concepts is usually large compared to the number of disjunctive clauses. As a result, these heuristics will often discover that all disjuncts have similar or equal priorities, and the guidance they provide is not particularly useful.

An alternative strategy is to employ an *oldest-first* heuristic that tries to maximise the effectiveness of backjumping by using dependency sets to guide the expansion [Horrocks and Patel-Schneider, 1999]. When choosing a disjunct on which to branch, the heuristic first selects those disjunctions that depend on the least recent branching points (i.e., those with minimal maximum values in their dependency sets), and then selects a disjunct from one of these disjunctions. This can be combined with the use of a BCP maximising heuristic, such as the Jeroslow and Wang heuristic, to select the disjunct from amongst the selected disjunctions.

Although the BCP and backjumping maximising heuristics described above have been designed with semantic branching in mind they can also be used with syntactic branching. The oldest first heuristic actually selects disjunctions rather than disjuncts, and is thus a natural candidate for a syntactic branching heuristic. BCP maximising heuristics could also be adapted for use with syntactic branching, for example by first evaluating the weighting of each disjunct and then selecting the disjunction whose disjuncts have the highest average, median or maximum weightings.

The oldest first heuristic can also be used to advantage when selecting the order in which existential role restrictions, and the labels of the R-successors which they generate, are expanded. One possible technique is to use the heuristic to select an unexpanded existential role restriction $\exists R.C$ from the label of a node x, apply the $\exists$-rule and the $\forall$-rule as necessary, and expand the label of resulting R-successor. If the expansion results in a clash, then the algorithm will backtrack; if it does not, then continue selecting and expanding existential role restrictions from $\mathcal{L}(x)$ until it is fully expanded. A better technique is to first apply the $\exists$-rule and the $\forall$-rule exhaustively, creating a set of successor nodes. The order in which to expand these successors can then be based on the minimal maximum values in the dependency sets of all the concepts in their label, some of which may be due to universal role restrictions in $\mathcal{L}(x)$.

The advantages of using heuristics are

- They can be used to complement other optimisations. The MOMS and Jeroslow and Wang heuristics, for example, are designed to increase the effectiveness of BCP while the oldest first heuristic is designed to increase the effectiveness of backjumping.
- They can be selected and tuned to take advantage of the kinds of problem that are to be solved (if this is known). The BCP maximisation heuristics, for example, are generally quite effective with large randomly generated and hand crafted problems, whereas the oldest first heuristic seems to be more effective when classifying realistic KBs.

The disadvantages are

- They can add a significant overhead as the heuristic function may be expensive to evaluate and may need to be reevaluated at each branching point.
- They may not improve performance, and may significantly degrade it.
  - Heuristics can interact adversely with other optimisations, as was the case with the MOMS heuristic and backjumping in the FaCT system.
  - When they work badly, heuristics can increase the frequency with which pathological worst cases can be expected to occur. For example, with problems that are highly disjunctive but relatively under-constrained, using a BCP maximising heuristic to select highly constraining disjuncts can force backtracking search to be performed when most random branching choices would lead rapidly to a clash free expansion.
  - The cost of computing the heuristic function can outweigh the benefit (if any).
- Heuristics designed to work well with purely proposition reasoning, such as the BCP maximising heuristics, may not be particularly effective with DLs, where much of the reasoning is modal (it involves roles and sub-problems). There has been little work on finding good heuristics for modal reasoning problems.

9.5.4.5 Caching satisfiability status

During a satisfiability check there may be many successor nodes created. Some of these nodes can be very similar, particularly as the labels of the R-successors for a node x each contain the same concepts derived from the universal role restrictions in $\mathcal{L}(x)$. Considerable time can thus be spent re-computing the satisfiability of nodes that have the same label. As the satisfiability algorithm only needs to know whether a node is satisfiable or not, this time is wasted. Moreover, when classifying a KB, similar satisfiability tests may be performed many times, and may provide further opportunities for the re-use of satisfiability results for node labels if these are retained across multiple concept satisfiability tests.

If the expansion of existential value restrictions in the label of a node x is delayed until all other expansion possibilities have been exhausted (as in the trace technique), then as each existential role restriction $\exists R.C$ is expanded it is possible to generate the complete set of concepts that constitute the initial label of the R-successor; this will consist of C plus all the concepts derived from universal role restrictions in $\mathcal{L}(x)$.[1] If there exists another node with the same set of initial concepts, then the two nodes will have the same satisfiability status. Work need be done only on one of the two nodes, potentially saving a considerable amount of

[1] This ordering is used in the trace technique to minimise space usage, and may by useful or even required for effective blocking.

processing, as not only is the work at one of the nodes saved, but also the work at any of the successors of this node.

Care must be taken when using caching in conjunction with blocking as the satisfiability status of blocked nodes is not completely determined but is simply taken to be equal to that of the blocking node. Another problem with caching is that the dependency information required for backjumping cannot be effectively calculated for nodes that are found to be unsatisfiable as a result of a cache lookup. Although the set of concepts in the initial label of such a node is the same as that of the expanded node whose (un)satisfiability status has been cached, the dependency sets attached to the concepts that made up the two labels may not be the same. However, a weaker form of backjumping can still be performed by taking the dependency set of the unsatisfiable node to be the union of the dependency sets from the concepts in its label.

A general procedure for using caching when expanding a node x can be described as follows.

(i) Exhaustively perform all local expansions, backtracking as required, until only existential value restrictions (if any) remain to be expanded.

(ii) If there are no unexpanded existential value restrictions in $\mathcal{L}(x)$, then return the satisfiability status *satisfiable* to the predecessor node.

(iii) Select (heuristically) an unexpanded existential role restriction from $\mathcal{L}(x)$, expanding it and any applicable universal role restrictions to create a new node y with an initial label $\mathcal{L}(y)$ (or create all such nodes and heuristically select the order in which they are to be examined).

(iv) If y is blocked, then its satisfiability status S is directly determined by the algorithm (normally *satisfiable*, but may depend on the kind of cycle that has been detected [Baader, 1991]).

  (a) If $S =$ *satisfiable*, then return to step (ii) without expanding $\mathcal{L}(y)$.

  (b) If $S =$ *unsatisfiable*, then backtrack without expanding $\mathcal{L}(y)$. The dependency set will need to be determined by the blocking algorithm.

(v) If a set equal to $\mathcal{L}(y)$ is found in the cache, then retrieve the associated satisfiability status S (this is called a cache "hit").

  (a) If $S =$ *satisfiable*, then return to step (ii) without expanding $\mathcal{L}(y)$.

  (b) If $S =$ *unsatisfiable*, then backtrack without expanding $\mathcal{L}(y)$, taking the dependency set to be the union of the dependency sets attached to the concepts in $\mathcal{L}(y)$.

(vi) If a set equal to $\mathcal{L}(y)$ is not found in the cache, then set $L = \mathcal{L}(y)$ and expand $\mathcal{L}(y)$ in order to determine its satisfiability status S.

(a) If $S =$ *satisfiable* and there is no descendent z of y that is blocked by an ancestor x' of y, then add L to the cache with satisfiability status S and return to step (ii).
(b) If $S =$ *satisfiable* and there is a descendent z of y that is blocked by an ancestor x' of y, then return to step (ii) *without* updating the cache.
(c) If $S =$ *unsatisfiable*, then add L to the cache with satisfiability status S and backtrack, taking the dependency set to be the one returned by the expansion of $\mathcal{L}(y)$.

The problem of combining caching and blocking can be dealt with in a more sophisticated way by allowing the cached satisfiability status of a node to assume values such as "*unknown*". These values can be updated as the expansion progresses and the satisfiability status of blocking nodes is determined. Such a strategy is implemented in the DLP system.

A further refinement is to use subset and superset instead of equality when retreiving satisfiability status from the cache: if $\mathcal{L}(x)$ is satisfiable, then clearly any $\mathcal{L}(y) \subseteq \mathcal{L}(x)$ is also satisfiable, and if $\mathcal{L}(x)$ is unsatisfiable, then clearly any $\mathcal{L}(y) \supseteq \mathcal{L}(x)$ is also unsatisfiable [Hoffmann and Koehler, 1999; Giunchiglia and Tacchella, 2000]. However, using sub and supersets significantly increases the complexity of the cache, and it is not yet clear if the performance cost of this added complexity will be justified by the possible increase in cache hits.

The advantages of caching the satisfiability status are:

- It can be highly effective with some problems, particularly those with a repetitive structure. For example, the DLP system has been used to demonstrate that some of the problem sets from the Tableaux'98 benchmark suite are trivial when caching is used (all problems were solved in less than 0.1s and there was little evidence of increasing difficulty with increasing problem size). Without caching, the same problems demonstrate a clearly exponential growth in solution time with increasing problem size, and the system was unable to solve the larger problems within the 100s time limit imposed in the test [Horrocks and Patel-Schneider, 1999].
- It can be effective with both single satisfiability tests and across multiple tests (as in KB classification).
- It can be effective with both satisfiable and unsatisfiable problems, unlike many other optimisation techniques that are primarily aimed at speeding up the detection of unsatisfiability.

The disadvantages are:

- Retaining node labels and their satisfiability status throughout a satisfiability

test (or longer, if the results are to be used in later satisfiability tests) involves a storage overhead. As the maximum number of different possible node labels is exponential in the number of different concepts, this overhead could be prohibitive, and it may be necessary to implement a mechanism for clearing some or all of the cache. However, experiments with caching in the DLP system suggest that this is unlikely to be a problem in realistic applications [Horrocks and Patel-Schneider, 1999].

- The adverse interaction with dependency directed backtracking can degrade performance in some circumstances.
- Its effectiveness is problem dependent, and (as might be expected) is most evident with artificial problems having a repetitive structure. It is highly effective with some of the hand crafted problems from the Tableaux'98 benchmark suite, it is less effective with realistic classification problems, and it is almost completely ineffective with randomly generated problems [Horrocks and Patel-Schneider, 1999].
- The technique described depends on the logic having the property that the satisfiability of a node is completely determined by its initial label set. Extend the technique to logics that do not have this property, for example those which support inverse roles, may involve a considerable increase in both complexity and storage requirements.

9.6 Discussion

To be useful in realistic applications, DL systems need both expressive logics and fast reasoners. Procedures for deciding subsumption (or equivalently satisfiability) in such logics have discouragingly high worst-case complexities, normally exponential with respect to problem size. In spite of this, implemented DL systems have demonstrated that acceptable performance can be achieved with the kinds of problem that typically occur in realistic applications.

This performance has been achieved through the use of optimisation techniques, a wide variety of which have been studied in this chapter. These techniques can operate at every level of a DL system; they can simplify the KB, reduce the number of subsumption tests required to classify it, substitute tableaux subsumption tests with less costly tests, and reduce the size of the search space resulting from non-deterministic tableaux expansion. Amongst the most effective of these optimisations are absorption and backjumping; both have the desirable properties that they impose a very small additional overhead, can dramatically improve typical case performance, and hardly ever degrade performance (to any significant extent). Other widely applicable optimisations include enhanced traversal, normalisation, lazy unfolding, semantic branching and local simplification; their effects are less general and less dramatic, but they too impose low overheads and rarely degrade

performance. Various forms of caching can also be highly effective, but they do impose a significant additional overhead in terms of memory usage, and can sometimes degrade performance. Finally, heuristic techniques, at least those currently available, are not particularly effective and can often degrade performance.

Several exciting new application areas are opening up for very expressive DLs, in particular reasoning about DataBase schemata and queries, and providing reasoning support for the Semantic Web. These applications require logics even more expressive than those implemented in existing systems, in particular logics that include both inverse roles and number restrictions, as well as reasoning with general axioms. The challenge for DL implementors is to demonstrate that highly optimised reasoners can provide acceptable performance even for these logics. This may require the extension and refinement of existing techniques, or even the development of completely new ones.

One promising possibility is to use a more sophisticated form of dependency directed backtracking, called *dynamic backtracking* [Ginsberg, 1993], that preserves as much work as possible while backtracking to the source of a contradiction. Another useful approach, indicative of the increasing maturity of existing implementations, is to focus on problematical constructors and devise methods for dealing with them more efficiently. Good examples of this can be seen in the RACER system, where significant improvements in performance have been achieved by using more sophisticated techniques to deal with domain and range constraints on roles (see Chapter 2 for an explanation of these constructs) and qualified number restrictions [Haarslev and Möller, 2001c; 2001d; 2001a].

Finally, it should be reemphasised that, given the immutability of theoretical complexity, no (complete) implementation can guarantee to provide good performance in all cases. The objective of optimised implementations is to provide acceptable performance in typical applications and, as the definition of “acceptable” and “typical” will always be application dependent, their effectiveness can only be assessed by empirical testing. Hopefully, the new generation of highly optimised DL systems will demonstrate their effectiveness by finding more widespread use in applications than did their predecessors.