View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Technische Universität Dresden: Qucosa

To and Fro
Between Tableaus and Automata for Description Logics

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inform. Jan Hladik
geboren am 7. Oktober 1972 in Bad Godesberg

Gutachter:

Prof. Dr.-Ing. Franz Baader
Technische Universität Dresden

Prof. Dr. rer. nat. habil. Ulrike Sattler
University of Manchester

Prof. Dr. rer. nat. habil. Michael Thielscher
Technische Universität Dresden

Tag der Verteidigung:
14. November 2007

Dresden, im Februar 2008

Contents

List of Figures

Chapter 1

Introduction

Description Logics (DLs) are a family of knowledge representation languages with a well-defined logic-based semantics and practically usable inference algorithms, which enable DL systems to deduce implicit information about the domain of interest from the explicitly provided facts. The syntax of DLs is based on the notion of *concepts*, which represent classes of individuals like Human, Animal or Machine, and *roles*, which stand for relations between individuals like has-child or owns. A specific DL is characterised by the *constructors* it provides for generating complex concepts from primitive ones; e. g. using conjunction, negation, and existential quantification, we can express the notion of "mother" as

$$\mathsf{Human} \sqcap \neg\mathsf{Male} \sqcap \exists\mathsf{has\text{-}child}.\mathsf{Human} \tag{1.1}$$

i. e. a human being who is not male and has a child who is also a human being.

The semantics of DL expressions is defined in a set-theoretic way: concepts and roles are interpreted as sets of individuals and sets of pairs of individuals, respectively, where the semantics of complex concepts depends on the semantics of the primitive concepts appearing in it, e. g. the concept $\mathsf{Human} \sqcap \neg\mathsf{Male}$ is interpreted as all the individuals from the interpretation of Human that do not appear in the interpretation of Male. The standard DL inference problems are the *satisfiability* problem, i. e. the question if a concept C can be interpreted as a non-empty set, and the *subsumption* problem, i. e. the question if every individual in the interpretation of a concept C necessarily belongs to the interpretation of another concept D.

Since in the presence of conjunction and negation it is possible to reduce subsumption to satisfiability, most DL reasoners have at their core a satisfiability tester; thus we will focus on the satisfiability problem in the following. In the area of description logics, one of the most widely used methods for deciding the satisfiability problem is the *tableau* approach, which is also the basis for the majority of current implementations. In order to test the satisfiability of an input, a tableau algorithm (TA) tries to generate a tree-shaped (pre-)model, called tableau, by breaking the syntactic structure of the input down thus going from the (complex) input problem to easier sub-problems. For the "mother" example above, a TA starts by creating a root node labelled with the entire concept term, then it adds the single conjuncts to the node

label, and finally it creates a successor node labelled with Human that is connected with the root node by an edge labelled with has-child. If all relevant subconcepts have been processed and the tree does not contain any obvious contradictions, this implies that the input is satisfiable.

In addition to soundness and completeness, *termination* of the TA must be ensured in order to obtain a decision procedure. For logics having the *finite tree model property*, the corresponding TA usually terminates "by itself", i. e. without the introduction of mechanisms with the sole purpose of guaranteeing termination. If additionally the tree models have polynomial depth and the different branches are independent in the sense that the possible labels for a node only depend on the label of its predecessors and successors, but not on that of other nodes within the tree, then the tableau algorithm can be shown to require only space polynomial in the size of the input by keeping only one branch in memory at a time.

For logics without the finite tree model property, TAs require a cycle detection mechanism in order to avoid constructing an infinite tableau. This is done by defining a *blocking* condition: a node is said to be blocked if it can be replaced by (a copy of) a predecessor node. Termination can then be ensured by avoiding the creation of successors for blocked nodes. Regarding complexity in this case, it turns out that the tableau algorithm usually requires exponential time in the worst case because it may be necessary to create an exponential number of nodes before one of them is blocked. Moreover, since TAs are usually non-deterministic (e. g. for logics with disjunction), the complexity classes that can be obtained "naturally" from a TA are non-deterministic, e. g. NExpTime.

In spite of this high worst-case complexity, implementations based on tableau algorithms perform surprisingly well on knowledge bases originating from real-life applications (see Section 2.3.7). The reasons for this behaviour are the following:

- Tableau algorithms are *goal-directed*: they only add new information to the tree if it is likely to lead the tree closer to a tableau. They do not randomly add concepts which may be irrelevant for the satisfiability of the input concept.
- Tableau algorithms are amenable to several well-known and efficient optimisations. Although a naive implementation may not show an acceptable performance, these optimisations can improve the speed by orders of magnitude (see Section 2.3.7).
- Knowledge bases resulting from real-life applications often do not make use of the full expressive power provided by the system; in other words, the kind of concepts occurring in an ExpTime-hardness proof rarely appears in "realistic" knowledge bases. Together with the goal-direction characteristic mentioned above, this allows the TA for an expressive DL to perform as well as a DL for an inexpressive language on an "easy" input.[1]

However, from a theoretical point of view, tableaus have several drawbacks, which were briefly sketched above and are described in more detail in the following.

[1] Other knowledge bases, e. g. ones resulting from automatic translation procedures, often do not satisfy these properties. Section 5.2.3 shows an example for this unfavourable behaviour.

- Tableau algorithms are usually non-deterministic, which is caused e. g. by the necessity to handle disjunction, thus giving rise to non-deterministic complexity classes. Regarding space complexity, this is not a problem since, by a result of Savitch (1970), the non-deterministic space complexity classes coincide with the deterministic ones,[2] and thus TAs are well-suited for obtaining worst-case optimal results for PSpace-complete logics (see Section 2.3.5). However, many expressive DLs are ExpTime-complete (see Section 2.3.6), and the upper bound that can be obtained from a TA "naturally", i. e. without tuning the algorithm in order to improve time efficiency, is only NExpTime (or even 2-NExpTime, see Section 3.2.2).

  Although there exist approaches to define tableaus with ExpTime worst-case complexity, the techniques necessary for achieving this result introduce a significant overhead in the algorithm, which makes the usability in practice questionable and, to the best of our knowledge, no implementation realising an ExpTime upper bound exists so far. (This issue is discussed in more detail in Section 3.2.2.) In contrast, the recently developed *hypertableau calculus* (see also Section 1.3) has been successful in avoiding non-deterministic rules and thus obtaining an algorithm that requires deterministic exponential time in the worst case and also performs well in practice, albeit only for knowledge bases having specific characteristics.

- Since tableau algorithms try to generate a tree-shaped model, they only terminate "naturally" for logics that have the finite tree model property. For other logics, termination has to be ensured by detecting repetition patterns in the generated model, i. e. by preventing the generation of nodes which are identical (or in some other way compatible) to other nodes that have already been generated. This technique, called *blocking*, often makes the soundness proof of the algorithm rather intricate. (Again, see Section 3.2.2 for a more detailed discussion of blocking.)

A different class of decision procedures for the satisfiability problem without the disadvantages of TAs is constituted by *automata*-based reasoning algorithms (AAs), which translate a DL input either into a non-deterministic tree automaton (NTA) (see Section 4.3) or into an alternating tree automaton (ATA) (Section 4.4). Satisfiability of the input expression can then be reduced to non-emptiness of the language accepted by the corresponding automaton. In the NTA case, the translation usually yields an automaton of exponential size for which emptiness can be tested in polynomial time, thus giving rise to an ExpTime complexity result. In the ATA case, the size of the automaton is usually polynomial in the size of the input, but the emptiness test requires exponential time (it implies a translation into an exponentially large NTA), consequently it also leads to an ExpTime result. Automata algorithms have the following advantages in comparison to tableaus:

[2] Since Savitch's algorithm involves a quadratic blow-up of the required space, this statement only holds for the classes PSpace and above, but this covers the space complexity classes for the logics considered in this thesis.

- The translation into an automaton as well as the automata emptiness test is deterministic. Therefore, the "natural" complexity class that can be obtained from an AA is also deterministic (ExpTime), which makes them better suited for achieving tight upper complexity bounds for ExpTime-complete logics.
- Since the emptiness test is performed on the (finite) automaton rather than a (possibly infinite) model, the algorithm terminates automatically without the need for a blocking condition. This makes an AA more elegant than the corresponding TA, and it significantly simplifies the proofs of soundness and completeness of the decision procedure (see Sections 3.2.2, 4.3.1, and 4.4.1).

However, these advantages come with a price. Automata algorithms have structural disadvantages in comparison with tableau algorithms:

- Both the NTA and the ATA approach involve an exponential step. Thus automata, unlike tableaus, cannot easily be used to prove a tight upper complexity bound for DLs in a lower complexity class, e. g. PSpace.
- Since, in the naive approach sketched above, the exponentially large automaton has to be generated before the emptiness test is performed, the automata algorithm will consume exponential time not only in the worst case, but in any case, thus a direct implementation of the automata approach does not promise acceptable performance in applications.

This problem cannot easily be circumvented by optimisations because the automata emptiness test starts with the set of all states and then iteratively eliminates states that cannot occur in a successful run of the automaton, which in turn depends on the other states that still remain (see Section 4.1 for details). Therefore, it is difficult to recognise "unnecessary" states in advance and avoid their generation. For the "mother" concept in Formula 1.1, the NTA generated by the automata algorithm described in detail in Section 4.3.1 has 14 states (see Page 52), only one of which is required to verify the existence of a successful run. This illustrates the overhead introduced by the automata approach.

1.1 Aim of This Thesis

The complementary advantages and disadvantages of the two decision procedures described above often make it necessary to develop two algorithms in order to introduce a new DL: an automata algorithm showing the ExpTime upper bound, and a tableau algorithm promising to lead to acceptable performance in an implementation. The necessity of this double effort is particularly displeasing because, from a higher level of abstraction, tableaus and automata become similar again: the transitions of the automaton *somehow* look like tableau rules, and a tableau generated by a TA *somehow* looks like an input accepted by the automaton arising from the AA; a similarity which is particularly strong in the case of alternating automata (see Section 4.4.1 for details). Consequently, the proofs for TAs and AAs are usually also based on the same ideas and observations.

However, this superficial similarity does not imply that it is easy to combine the advantages because, upon closer examination of the details of the two approaches, it becomes clear that their different properties result from the fact that the algorithms work in opposite directions: TAs go from complex to simple problems, which allows them to only consider subproblems relevant for the input, but this technique makes backtracking necessary for non-deterministic decisions. A TA can therefore be regarded as a *non-deterministic top-down* procedure. On the other hand, an automata algorithm is a *deterministic bottom-up* procedure: it starts by considering simple problems (more precisely, types whose satisfiability does not depend on the satisfiability of other types) and deduces the answer to complicated problems from these. The bottom-up approach thus allows an AA to be deterministic, but it is (in a non-optimised version) not goal-directed and will therefore compute a significant amount of information that is irrelevant for the input problem, whereas the TA can restrict attention to relevant information, but this implies that it has to non-deterministically guess the right path to take.

The aim of this thesis is to attempt to reconcile the advantages of the two approaches in spite of these obstacles. In particular, we are trying to answer the following questions:

1. What are the precise relations between the data structures used in the different approaches? Can a tableau constructed by a TA serve as an input for the corresponding AA?

2. Is it possible to achieve acceptable performance in practice with an automata algorithm using techniques stemming from tableau algorithms?

3. Is it possible to transfer the complexity results in either direction, i. e. is it possible to obtain a PSpace result from an AA or an ExpTime result from a TA?

In order to answer these questions, we aim at obtaining a clear formalisation of the similarities that are only vaguely sketched above and to use these observations in order to transfer techniques and desirable properties between the two approaches.

1.2 Outline of This Thesis

In Chapter 2, we begin with formally introducing DLs and outlining the properties of those logics that will be relevant in the remainder of the thesis. Chapter 3 describes the origin of semantic tableaus for classical logic and the adaptation of this technique for DLs. Similarly, Chapter 4 gives the relevant background of automata theory, describing the two automata models which are particularly useful for DLs, namely *non-deterministic tree automata* and *alternating two-way tree automata*.

In Chapter 5, we describe our first approach to transfer practical efficiency from tableaus to automata by translating an automaton into a DL knowledge base, which allows us to perform the automata emptiness test with a tableau-based reasoner. Although the performance of this approach does not turn out to be satisfactory, it

emphasises the close relationship between the operation of tableaus and automata algorithms as well as between the data structures used by the two methods. As a side product, we gain a new ExpTime complexity result for the inexpressive DL $\mathcal{ELU}_f$, which is used for the translation.

Since this first attempt does not lead to an acceptable behaviour in practice, our next aim is to improve the theoretical efficiency for the automata approach and to lower the complexity class obtainable through an AA from ExpTime to PSpace by adapting the blocking technique from tableaus. In Chapter 6, we develop two frameworks for automata algorithms for PSpace logics: the first one is designed for logics with the finite tree model property and comparably easy to use, whereas the second one is more general, but requires more work from the user. With the help of the more powerful framework, we are able to prove, using an automata algorithm, that the concept satisfiability problem with respect to acyclic TBoxes (see Section 2.3.2) for the DL $\mathcal{SI}$ is in PSpace.

In Chapter 7, we go in the opposite direction and investigate conditions under which ExpTime complexity results can be transferred from automata to tableau algorithms. For this purpose, we define the framework of *tableau systems*, which provides a formal notion of tableau algorithms for ExpTime logics. From a tableau system, we obtain an ExpTime automata algorithm and a terminating tableau algorithm that is amenable to the well-known optimisations for TAs and likely to exhibit good performance in practice. To achieve this goal, it is not necessary to define a blocking condition; instead, an appropriate condition is derived from the parameters of the tableau system. By defining a system for the DL $\mathcal{SHIO}$, we illustrate the usefulness of this framework and obtain a new ExpTime complexity result.

1.3 Related Work

To the best of our knowledge, no attempt has been made to achieve a practically useable reasoner by directly implementing an automata algorithm (thus answering Question 2 from the end of Section 1.1). However, there exist two approaches for testing satisfiability in the modal logic **K** (which is a notational variant of the DL $\mathcal{ALC}$, see Section 2.3) which are based on algorithms that can be regarded as optimised variants of the automata method:

- Baader and Tobies (2001) show that the so-called inverse method (Voronkov, 2001) for deciding satisfiability for **K** formulas can be viewed as an implementation of the automata-based approach, which suggests that it could also be useful for DLs. The implementation Kk (Voronkov, 1999) performed well in practice (Voronkov, 2001), but to the best of our knowledge, this reasoner has not been extended to more expressive logics, and no results have been published recently.

- The satisfiability algorithm for **K** presented by Pan, Sattler, and Vardi (2006), which is based on *binary decision diagrams* (BDDs), can also be regarded as an optimised automata emptiness test. Like the inverse method, the BDD-based

approach exhibited good results for **K**, but it has not been extended to more expressive logics, and new results have not been published recently.[3]

Developing a tableau or automata algorithm is not the only method for obtaining a complexity result or a practical decision procedure for the satisfiability problem of a description logic. Some approaches use translation from the corresponding DL into other formalisms with known complexity results or existing reasoners, and other ones are based on the extension of reasoning mechanisms for less expressive formalisms like propositional logic. In the following, we briefly describe some of these methods.

- The modal logic reasoner MSpass (Hustadt and Schmidt, 2000), which can be regarded as a satisfiability tester for some DLs due to the relation between MLs and DLs, translates an expression from modal into predicate logic and uses the resolution-based reasoner Spass (Weidenbach, 1999; Weidenbach, Afshordel, Brahm, Cohrs, Engel, Keen, Theobalt, and Topić, 1999) to decide the satisfiability problem.

- The more recent DL reasoner Kaon2 (Hustadt, Motik, and Sattler, 2004, 2007) reduces a DL expression to a disjunctive datalog program, which allows the developers to apply the known optimisation techniques for deductive databases and thus to obtain a practically efficient ExpTime procedure (Motik and Sattler, 2006).

- The *hypertableau calculus* (Motik, Shearer, and Horrocks, 2007) can be regarded as a hybrid of resolution and and the tableau approach: here a DL knowledge base is translated into a clausal form, and reasoning is performed via resolution. If the result of the translation is a set of Horn clauses (which requires the knowledge base to have a specific shape) then this calculus gives rise to a deterministic reasoning algorithm and thus to a (worst-case optimal) ExpTime result.

- Translation from DLs into converse **PDL** (Schild, 1991; De Giacomo and Lenzerini, 1994a,b) allows for the transfer of complexity results (e. g. that $\mathcal{ALC}$ augmented with transitive-reflexive closure of roles is ExpTime-hard) and model theoretic properties (e. g. that this DL augmented with role union, composition and inverses has the finite model property).

- The reasoner Ksat (Giunchiglia and Sebastiani, 1996; Giunchiglia, Giunchiglia, Sebastiani, and Tacchella, 1998) uses a method based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm (Davis and Putnam, 1960; Davis, Logemann, and Loveland, 1962) for satisfiability testing in propositional logic in order to test the satisfiability of $\mathcal{ALC}$ concepts. Similar to the BDD algorithm and the inverse method mentioned above, the Ksat approach has not been extended to logics that are more expressive than $\mathcal{ALC}$, and no results have been published recently.

[3] Although the cited paper was published in 2006, the work it is based on was performed in 2002 (Pan, Sattler, and Vardi, 2002) and 2003 (Pan and Vardi, 2003).

Translation into other formalisms has the clear advantage of allowing for the use of optimised systems that are available for the target formalism. However, the limitations are equally obvious: unlike for tableaus, where an existing algorithm can be extended to a more expressive language with additional constructors by introducing new tableau rules (and possibly modifying the existing ones), a translation approach can only be extended if the additional constructors can also be expressed in the target formalism.

Chapter 2

Knowledge Representation with Description Logics

In this chapter, we will first briefly recall some of the relevant philosophical discussion regarding the term *knowledge*; we will then describe how the computer science discipline of *knowledge representation* endeavours to store and retrieve knowledge with the help of computers, and we mention some of the early attempts to achieve this goal and their deficiencies, which eventually led to the introduction of *description logics*. We describe in detail description logics of different expressivity, the formalisms that they use to represent knowledge, and the mechanisms that they provide to make implicit knowledge explicit. Finally, we give a brief overview of description logic systems.

2.1 Knowledge

"What, in your opinion, is knowledge?" (Plato, Theaetetus, 146b). The discussion about a proper definition goes back to ancient Greece, more precisely to Plato's *Theaeteus*. Although this dialogue, like most of Plato's early works, ends with the conclusion that a satisfactory answer cannot be given (and thus, as hinted in the initial question, we do not obtain knowledge about the idea of knowledge, but only opinions), several necessary conditions for a definition of knowledge are found. For a person P to know a fact F,

1. F must be *true*,
2. P must *believe* F,
3. P must have a *justification* for F.

The first two conditions are easy to accept as necessary for knowledge: it is impossible to know something without assuming that it is true; and the truth of F is exactly what marks the difference between knowledge and mere belief or opinion. Still, it is possible to believe a true fact by coincidence without knowing it, and therefore the third condition also is necessary. This one, however, shows in Plato's opinion that

knowledge is impossible: since knowledge can only be achieved analytically, i. e. by splitting a complex problem into several simpler ones, one either ends up in an infinite analytical process or with atomic problems, which cannot be analysed. Consequently, no knowledge about these problems (and thus also about the more complex ones) is possible. This dead end, sometimes called the *Münchhausen trilemma* (Albert, 1991), leaves only three alternatives: *infinite regression*, where one continuously passes from a justification to the justification for the justification, and consequently sinks deeper and deeper into the "swamp"; a *circular argument*, where one finds that the justification for a fact is founded on the fact itself, and one therefore remains stuck at the same place; and finally *dogmatism*, where one simply declares some fundamental facts to be true and thus pulls oneself out of the swamp by his own hair. Like in the tale, the third way, though it involves "cheating", is the only viable one. For example, Aristotle does not claim to prove the law of non-contradiction (Aristotle, Metaphysics, 1005b), but he only argues that without it, a reasonable discussion is impossible, and subsequently takes it as granted "for it shows a lack of education not to know of what things one should demand demonstration, and of what one should not" (Aristotle, Metaphysics, 1006a). Similarly, in mathematics a set of axioms is assumed to be true, and one is only interested in the truth of a theorem in the theory induced by these axioms.

Another problem with the definition of knowledge as *justified true belief* is the vague notion of justification. Clearly, a true belief can also be justified by, e. g., prejudice or superstition, without being knowledge; an imprecision which has to be attributed to the fact that Plato does not yet have a formal notion of the term "proof". Aristotle developed such a notion with the syllogisms: a proof is "an argument in which, certain things having been supposed, something different from the supposed things results of necessity because of their being so" (Aristotle, Prior Analytics, 24b). With this, he was able to formulate stricter demands for justifications of knowledge: "we believe that we know a single thing absolutely—and not in the sophistic way, which is casual—whenever we believe to be aware of the reason by which the thing is, that it is the reason of the thing, and that it does not allow the thing to be different." (Aristotle, Posterior Analytics, 71b). Thus, knowledge of a thing involves the awareness of a *sufficient reason*, which rules out vague justifications. In mathematics, this claim is reflected in the demand for *soundness* of a proof, i. e. the condition that for every step in the argument, truth of the premisses implies truth of the consequence.

Within this thesis, we cannot attempt to present an detailed survey of the philosophical discussion about the term "knowledge". However, even this brief introduction has revealed the fundamental assumptions that are still present in its contemporary notion: we have established that knowing a fact F means believing F, being aware of the axioms F depends on, and how these axioms necessarily entail F, which implies that F is true if the axioms are true. Transferring this notion from humans to machines, this means that for a system S to represent the knowledge of a fact F, it is necessary that

- F is true with respect to the given axioms and the semantics of S,
- S has a proof for F, i. e. S can derive F from the axioms.

Thus, if the calculus of the system S is sound (i. e. it only derives true consequences), complete (i. e. it can derive all true consequences), and terminating (i. e. does not run infinitely), then S represents the knowledge of everything that is true with respect to the given axioms.

2.2 Knowledge Representation

The discipline of knowledge representation (KR) is an area of artificial intelligence that is concerned with computer-based storage and retrieval of knowledge about a part of the world, called the *domain of interest* (Lakemeyer and Nebel, 1994; Owsnicki-Klewe, von Luck, and Nebel, 1995). The immediate question is when it is justified to say that a computer system *represents knowledge*. Certainly, in a very general sense, every working program implicitly represents the knowledge necessary to perform its specific task, but if this knowledge is not stored explicitly in a human-readable form, it is hidden from the user and cannot be inspected, modified or extended by him.

Therefore, the *knowledge representation hypothesis*, as formulated by Smith (1982), demands two properties from a knowledge representation system: firstly, the knowledge has to be stored in a form that, *for an external observer*, represents the knowledge exhibited by the system; and secondly, it has to play *a causal and essential role* in the behaviour of the system. Without the first demand, as argued above, any software system could be regarded as representing knowledge; without the second one, i. e. if the represented knowledge did not have any influence on the system's behaviour, the system would not be guaranteed to behave according to the knowledge. As an example, Levesque and Brachman (1985) mention knowledge contained in comments of a program's source code; a more recent example could be an online encyclopedia that contains a huge amount of knowledge, but since it is not given in a machine readable form, it is impossible for the system to answer questions regarding the content, to derive new knowledge, or to detect contradictions. Thus, in a nutshell, knowledge representation requires the knowledge to be represented in a form that is *intelligible by both humans and machines*.

Consequently, a *KR system* consists of two parts: the *knowledge base (KB)*, which contains the *axioms* that the system knows about the domain, and the *inference engine*, which derives new facts from the axioms and provides the proofs that are required in order to represent knowledge. Thus, unlike a database, a KR system makes the implicit knowledge explicit (Baader, 1999), i. e. it does not just return the axioms that were previously entered, but it presents to the user the *theory induced by the axioms* (Owsnicki-Klewe et al., 1995). In the words of Brachman and Levesque (2004), the inference engine bridges the gap between the (necessarily finite) set of explicitly stored axioms and the (possibly infinite) knowledge represented by the system. In another paper (Levesque and Brachman, 1985), the same authors point out that the expressive power of a KR formalism (in this case, predicate logic) *determines not so much what can be said, but what can be left unsaid.* In many KR formalisms, it is easy to express an infinite structure with few axioms. By contrast, in a database nothing

that should be returned to the user can be left unsaid, and thus databases cannot be regarded as KR systems.

The separation of knowledge base and inference engine is one of the main features of KR systems. It enables the *knowledge engineer*, i. e. the person concerned with formalising the knowledge about the domain, to work independently of the *programmer*, i. e. the person implementing the inference algorithms. The programmer thus does not have to be an expert about the domain of interest, and the knowledge engineer does not require experience in programming. It is, however, necessary for him to understand how to express his knowledge in the *KR language*, the machine-readable formalism of the knowledge base. The features of the KR language and of the inference engine determine the properties of the KR system: the expressive power of the KR language determines what can be said in the system, the set of provided inference tasks determines which kinds of questions the system can answer, and the complexity of the reasoning tasks for the specific language determine how expensive (in terms of processing time and memory used) a query is. Clearly, there is a tradeoff between expressivity and efficiency (Levesque and Brachman, 1985): an inexpressive language may prevent the knowledge engineer from adequately describing the domain, whereas a very expressive one may lead to a high computational complexity of the reasoning tasks or even to undecidability.

The *semantics* of the language determines which conclusions should be derived by the inference engine or, in the words of the previous section, what is *true* for the KR system. This semantics can be *operational*, i. e. defined by means of the procedures that perform the reasoning tasks, or *declarative*, i. e. depending only on the represented knowledge (Baader, 1999; Russell and Norvig, 2002). Only a declarative semantics allows for a clear separation between the knowledge base and the inference engine, which is desirable for several reasons:

- A declarative semantics is necessary for a clear separation between the tasks of the knowledge engineer and the programmer. With an operational semantics, the knowledge engineer is required to know about the inference algorithms, and it may be necessary to modify the knowledge base if the algorithms change.
- It should be possible to use different inference engines (e. g. from different programmers, or optimised versions by the same programmer) on the same knowledge base while still obtaining the same results.
- Syntactic variants of expressing the same knowledge should not lead to different results. With an operational semantics, it is difficult to ensure that two syntactic variants of expressing the same knowledge lead to the same results (it is even difficult to use the term "syntactic variant", since the behaviour of the algorithm only depends on the syntax).

Already in the beginning of the AI era, McCarthy (1958) realised some of the advantages of a declarative semantics, in particular the independence of syntactic variants and of previous states of the program. He thus constrained that his AI project, called the *Advice Taker*, would be instructed with declarative, as opposed to instructive, sentences.

Semantic Networks. An early KR system that did not have a declarative semantics are *semantic networks* (Quillian, 1967). A semantic network is essentially a graph, where nodes stand for individuals, classes of individuals, or properties. Edges can either be *IS-A* links, which state that an individual is an element of a class or that a class is a subclass of another one, or *Has-Prop* links, which assign a property to a class or individual. Properties are *inherited* along IS-A links: if Spike IS-A Dog and Dog Has-Prop has-ears, then Spike Has-Prop has-ears.

The syntax of semantic nets is (seemingly) very intuitive, because it allows for a representation that is very similar to ordinary language (e. g. "Spike IS-A Dog IS-A Mammal IS-A Animal"). The problem is that also ambiguity and imprecision of ordinary language are present in semantic nets: IS-A can stand for "is an element of" or "is a subclass of", and a property edge can stand for "always has the property", "usually has the property" or "has the property, if any"). Brachman (1983) distinguishes six different meanings for "subclass" links and four for "element" ones. Clearly, it is impossible to obtain predictable results from a KR system if one syntax is used with different semantics.

Another problem arises with the interpretation of IS-A links as providing *default* properties for the subclass, which can be overridden by more specific properties assigned to the subclass. This is used e. g. to express that ostriches are birds, birds can fly, but ostriches cannot. If a subclass can override any property of the superclass, however, then IS-A cannot be regarded as representing taxonomic knowledge anymore: it is possible to say, e. g., that a Quadrangle IS-A Triangle with four instead of (the default) three edges (see also Brachman, 1985), and thus every node can be placed anywhere in the IS-A hierarchy.

Besides inheritance, *spreading activation* (Quillian, 1967; Collins and Loftus, 1975) is another inference for semantic nets that, according to Quillian, is supposed to "discover various relationships between the meanings two words". In principle, it is a process that starts by marking the nodes representing the two words under consideration and then iteratively marks all nodes connected with a marked node until an intersection is found. The path between the two nodes is then transformed into a sentence and returned to the user as a description for the relation between the words. The number of nodes on this path is regarded as a measure of the *semantic distance* between two words (Brachman, 1979). The problems with such a syntactic approach are obvious: a "naive" network with an IS-A link from Dog to Animal would deduce that dogs and animals are closely related concepts, whereas a "scientific" network with nodes for all intermediate concepts (Canis, Canidae, Carnivora, Mammalia etc.) would conclude that they are only remotely related.

Frames. The more complex KR formalism of *frames* was developed by Minsky (1975). A frame is a data structure that is supposed to contain all the knowledge relevant for a stereotypical situation. For this purpose, a frame contains *slots* that can be assigned specific values (*fillers*), which can also be frames. In order to represent a specific situation, one *instantiates* a frame by assigning values to these slots. For example, a Person frame could contain slots for Name, Age and Job, where Name requires a text string as filler, Age requires an integer and Job refers to an instance

of the Job frame. A frame can be a sub-frame of another one (similar to the IS-A construct of semantic nets), it can provide *defaults* for its slots, which are used unless more specific information is available, and it can provide procedures describing how to obtain the corresponding information.

An important inference is *criteriality* (see e. g. Hayes, 1979): if fillers can be found for all slots, then it can be deduced that the current situation is really an instance of the corresponding frame, i. e. having these fillers is a sufficient condition for being an instance. In the example, if something has a name, an age and a job, then it must be a person. *Matching* (Bobrow and Winograd, 1977) is another inference that decides if an instance of one frame can also be seen as an instance of another frame. For instance, if there is a frame Child that describes a Person whose Age is less than 14, then one can try to match an instance of the Person frame against the child frame and, if the matching is successful, deduce that the person is a child.

Minsky does not provide a formal semantics for frames; instead, he argues quite strongly against any logical foundation and in particular against a strict separation of the knowledge base and the inference services (which he calls *propositions* and *syllogisms*). He claims (Minsky, 1975) that in order to simulate common-sense thinking,

- one has to go from an imperfect, flawed initial model of the world to a more refined, less faulty one;
- it is necessary to restrict the application of inferences depending on the current situation, thus the current data should determine which inferences to apply;
- one also has to allow for the representation of actions or assumptions that *usually* apply, i. e. defaults;
- the notion of always-correct assumptions makes it impossible to capture a relevant part the real world with an acceptable performance;
- soundness (called *consistency* by Minsky) is not necessary or even desirable, since humans do not behave consistently.

However, Hayes (1974) argues that without a semantic theory, it is impossible to say what a certain statement claims about the world. Conversely, the knowledge engineer cannot know if the statement he wrote expresses what he has in mind. Moreover, it is impossible to compare different representational formalisms, e. g. to demonstrate that one formalism can express properties that another one cannot. Regarding the need for soundness, McDermott (1986) stresses the importance of *understanding* the system that one develops. Without a formal semantics, it is impossible to determine if an unexpected answer of a KR system is the result of a faulty knowledge base, a programming error in the inference engine or just an imperfection of the current representation that may be repaired by further development. Like McDermott, Hayes (1979) points out that it is possible to express defaults in classical logic, and he refers to Reiter (1978) as an example for reasoning by default in a logic with formal semantics.

Hayes also emphasises the need to distinguish between representation languages and programming languages (and thus between knowledge bases and inference engines): if one allows knowledge to be represented by programming languages then, as

argued in the beginning of this section, this does not justify using the term knowledge representation, since the expressions do not carry meaning; moreover, all (Turing-complete) programming languages would be equally expressive. Additionally, Hayes shows how the inferences of criteriality and matching can be decided using an intuitive translation of frames into predicate logic, thus refuting Minsky's claim that logic was inherently incapable of capturing the capabilities of frames.

Hayes concludes with the observation that the power of frames is not in the representational, but in the implementational level; and considering that the *object-oriented programming* paradigm with the capabilities of inheritance, data members and member functions resembles Minsky's frames much more closely than present-day KR systems do, this statement appears to have been very appropriate.

2.3 Description Logics

In order to overcome the problems of the early KR formalisms described above, Brachman and Schmolze (1985) developed the *Knowledge Representation Language Number One*, KL-ONE, which has a declarative, logic-based semantics and is commonly regarded (Woods and Schmolze, 1990; Nardi and Brachman, 2003) as the ancestor of the family of *Description Logics, (DLs)* (Baader, Calvanese, McGuinness, Nardi, and Patel-Schneider, 2003a). Schmidt-Schauß (1989) discovered that KL-ONE is undecidable and defined the decidable language $\mathcal{ALC}$ (Schmidt-Schauß and Smolka, 1991). Shortly after the introduction of $\mathcal{ALC}$, Schild (1991) observed that it is in fact a syntactic variant of the multi-modal logic $\mathbf{K}_m$ (see e. g. Blackburn, de Rijke, and Venema, 2001), which led to the transfer of several extensions and the corresponding complexity results from modal to description logics (e. g. *inverse roles* in DLs correspond to *converse modalities* in modal logics).

In the early phase of DL research, an important goal was the development of logics with polynomial-time inference algorithms. However, Nebel (1990) showed that even very inexpressive logics do not satisfy this condition: for the DL $\mathcal{FL}_0$, which only allows for conjunction and universal quantification, reasoning with respect to acyclic terminologies is co-NP-complete. Moreover, the analogy between $\mathcal{ALC}$ and $\mathbf{K}_m$ implies PSPACE-completeness of basic reasoning tasks in $\mathcal{ALC}$. On the other hand, the development of systems performing well in practice in spite of deciding intractable problems (for details, see Section 2.3.7) showed that tractability is not an indispensable condition. (Levesque and Brachman (1985) note, in a general context, that "it might be the case that for a wide range of questions, the program behaves properly, even though it can be shown that there will always be short questions whose answers will not be returned for a very long time, if at all." This clearly applies to reasoning for expressive DLs.) Consequently, the expressivity of DLs has been increased continuously to cover quantitative constraints on models *(features, number restrictions, nominals)* or restrictions on roles *(transitivity, hierarchies, inverse of roles)*. Although these extensions lead to EXPTIME- (or even NEXPTIME-)completeness, the existing optimised algorithms still perform reasonably well on knowledge bases from real-life applications..

This efficiency in practice, together with the high expressivity of these DLs, recently has led to the acceptance of the language OWL-DL (Bechhofer, van Harmelen, Hendler, Horrocks, McGuinness, Patel-Schneider, and Stein, 2004), which is based on a NExpTime-complete DL, as a standard for the *Semantic Web*[1], an extension of the World Wide Web, whose goal is to allow software agents to perform reasoning based on semantic annotation of web pages.

In parallel with the effort to increase the expressive power while still maintaining decidability and usability "in practice", the research of DLs with polynomial inference algorithms has recently regained attention (see e. g. Brandt, 2004; Baader, Brandt, and Lutz, 2005), and the polynomial subsumption algorithm has performed well on existing very large knowledge bases (Baader, Lutz, and Suntisrivaraporn, 2007b).

Thus, in the 20 years since the development of KL-One, numerous decidable DLs have been developed, with a wide range of expressivity (see the following sections) and various inference services (consistency, subsumption (Baader, 1999), least common subsumer (Cohen, Borgida, and Hirsh, 1992; Baader, Küsters, and Molitor, 1999), approximation (Brandt, Küsters, and Turhan, 2002), matching (Baader, Brandt, and Küsters, 2001), conjunctive queries (Glimm, Horrocks, Lutz, and Sattler, 2007), conservative extensions (Ghilardi, Lutz, and Wolter, 2006)) whose computational complexity ranges from polynomial (e. g. subsumption in $\mathcal{EL}$ (Baader, 2003)) to NExpTime (satisfiability in $\mathcal{SHOIQ}$ (Horrocks and Sattler, 2005)) and 2-ExpTime (conservative extensions for $\mathcal{ALC}$ (Ghilardi et al., 2006)).

2.3.1 Syntax

The common core for all DLs are the notions of *concepts*, which stand for classes of individuals, e. g. Animal or Human, and *roles*, which stand for a relations between two individuals, e. g. has-part or eats. A specific DL is characterised by the constructors it provides for the generation of complex concept and role terms from primitive ones. Firstly, there are the Boolean constructors $\sqcup$ (or), $\sqcap$ (and) and $\neg$ (not) for concepts, which have the same meaning as in propositional logic. Using these, we can describe a male animal as $\mathsf{Animal} \sqcap \neg\mathsf{Female}$, and a concept for both mammals and fishes as $\mathsf{Mammal} \sqcup \mathsf{Fish}$. Many DLs also allow for the *top* concept $\top$ and the *bottom* concept $\bot$, which stand for "everything" and "nothing", respectively. In the presence of conjunction/disjunction and negation, they can be regarded as shortcuts for the concept terms $\mathsf{A} \sqcup \neg\mathsf{A}$ and $\mathsf{A} \sqcap \neg\mathsf{A}$ for an arbitrary concept name A.

Using the quantifiers $\forall$ and $\exists$, we can talk about the relations between individuals from one class with that of another one: an *existential restriction* $\exists\mathsf{r}.\mathsf{C}$ describes a class of individuals that are related via the role r to individuals belonging to the concept C. Thus, the class of carnivores can be described by $\mathsf{Animal} \sqcap \exists\mathsf{eats}.\mathsf{Meat}$. This means that every individual belonging to this class is related via the role eats to another individual from the class Meat, but there may also be other eats relations. In contrast, a *value restriction* $\forall\mathsf{r}.\mathsf{C}$ describes the class of individuals i for which it holds that every individual to which i is related via the role r belongs to the concept C. A

[1] See `www.semanticweb.org`

freshwater fish could be described as $\mathsf{Fish} \sqcap \forall \mathsf{lives\text{-}in}.(\mathsf{River} \sqcup \mathsf{Lake})$. This means that every location the individual lives in must be a lake or a river, but it leaves open the possibility that there is no location given. *Nominals* are concept names that only represent one single individual. This makes it possible to express that some concept can only have one instance (e. g. God in a monotheistic knowledge base), or to give names to individuals (e. g. Rome or John) and use these names in concept definitions (Roman or Friend-of-John).

It is also possible to construct complex roles from role names using role constructors: the *inverse* operator $\cdot^{-}$ uses a role in the opposite direction. If we describe a predator with $\mathsf{Animal} \sqcap \exists \mathsf{eats}.\mathsf{Animal}$, we can analogously describe a prey by $\mathsf{Animal} \sqcap \exists \mathsf{eats}^{-}.\mathsf{Animal}$, meaning that it is an animal eaten by an animal. *Qualifying number restrictions* denote the class of individuals i for that there exists a bound on the number of individuals of a certain class that i is related to via a certain role. A popular person could be described as $\mathsf{Human} \sqcap (\geqslant 10\ \mathsf{has\text{-}friend}\ \mathsf{Human})$, a human with at least ten friends. In order to capture transitivity of roles, there exist two important approaches (Sattler, 1996): the *transitive closure* operator $\cdot^{*}$ allows for the expression $\exists \mathsf{has\text{-}child}^{*}.\mathsf{Male}$, which describes someone who has a male descendant. In contrast, *transitive roles* only allow for a role to be globally declared as transitive, which means that it is impossible to distinguish between direct and indirect successors. For example, a woman who has only daughters as children but also one male descendant can be described using transitive closure (as $\exists \mathsf{has\text{-}child}^{*}.\mathsf{Male} \sqcap \forall \mathsf{has\text{-}child}.\neg\mathsf{Male}$), but not with transitive roles. This higher expressivity comes with a price: concept satisfiability in $\mathcal{ALC}$ with transitive roles is PSPACE-complete, but with transitive closure it becomes EXPTIME-complete (Sattler, 1996).

Role Hierarchies are another way to express constraints about roles; they allow to declare one role as a sub-role of another one, e. g. $\mathsf{has\text{-}child} \sqsubseteq \mathsf{has\text{-}descendant}$. *Features* are functional roles, i. e. they describe relations that can be fulfilled by at most one individual, e. g. has-mother. Intersection, union, negation and composition (i. e. chaining) of roles have also been considered (De Giacomo, 1995). Due to the relation between modal and description logics, $\mathcal{ALC}$ with transitive closure, union and composition of roles can be shown to be a syntactic variant of **PDL** (Schild, 1991).

2.3.2 Knowledge Bases

In a DL system, the knowledge base consists of two components: the *terminological knowledge* about the nomenclature of the domain of interest is contained in *TBoxes* and *RBoxes*, and the *assertional knowledge* about individuals is contained in *ABoxes*. A TBox contains information about the relation between concepts, which can be in the form of *concept definitions* or *general concept inclusion axioms (GCIs)*. A concept definition assigns a name to a concept term. It has the form $\mathsf{A} \doteq \mathsf{C}$, where A is a concept name and C is a concept term, meaning that every individual that belongs to the class A also belongs to C and vice versa; thus, if we want to talk about freshwater fish, we can define a concept $\mathsf{Freshwater\text{-}fish} \doteq \mathsf{Fish} \sqcap \forall \mathsf{lives\text{-}in}.(\mathsf{River} \sqcup \mathsf{Lake})$. A GCI expresses additional restrictions about the domain. It has the form $\mathsf{C} \sqsubseteq \mathsf{D}$, where both C and D are concept terms. The meaning is that every individual that belongs to the

class C also belongs to D, but both classes need not be equal. So we can say that all mammals have hearts, but they are not necessarily the only such animals, with the GCI $\mathsf{Mammal} \sqsubseteq \exists \mathsf{has\text{-}part}.\mathsf{Heart}$.

A concept definition $\mathsf{A} \doteq \mathsf{C}$ can easily be expressed by the two GCIs $\mathsf{A} \sqsubseteq \mathsf{C}$ and $\mathsf{C} \sqsubseteq \mathsf{A}$, and it is also possible to express a GCI $\mathsf{C} \sqsubseteq \mathsf{D}$ using a new concept name Z and the definition $\mathsf{Z} \doteq \neg\mathsf{Z} \sqcup (\neg\mathsf{C} \sqcup \mathsf{D})$. This shows that without any restrictions, concept definitions can express very strong conditions on the models instead of just introducing a short name for a complex concept, and thus there exist two versions of TBoxes: *general* TBoxes allow for arbitrary definitions and GCIs, whereas *acyclic* TBoxes do not permit GCIs, multiple definitions of the same concept, or recursive definitions. For many logics, it turns out that reasoning w. r. t. acyclic TBoxes is in a lower complexity class than reasoning w. r. t. general TBoxes, see e. g. Chapter 6.[2]

Analogously to GCIs, which express axioms about concepts, there also exist *role inclusion axioms (RIAs)* of the kind $\mathsf{r} \sqsubseteq \mathsf{s}$, meaning that r is a subrole of s, e. g. $\mathsf{has\text{-}child} \sqsubseteq \mathsf{has\text{-}descendant}$.[3] An RBox is a finite set of RIAs.

An ABox contains information about the concepts and roles that the single individuals of the domain of interest belong to. They have the form $\mathsf{a} : \mathsf{C}$ or $(\mathsf{a}, \mathsf{b}) : \mathsf{r}$ for individual names a and b, a concept C and a role r. Thus, we can say that Mary is human and has a child named Jack as $\{\mathsf{Mary} : \mathsf{Human}, (\mathsf{Mary}, \mathsf{Jack}) : \mathsf{has\text{-}child}\}$.

2.3.3 Semantics and Inferences

As mentioned above, a main motivation for the introduction of DLs was to overcome the problems of semantic networks and frames by using a declarative, set-theoretic semantics. Thus, the semantics is defined by using a set of individuals (the *interpretation domain*) and assigning subsets of the interpretation domain to concepts and binary relations over the domain to roles. The standard inference problems for DL reasoning are satisfiability ("Is it possible to assign a non-empty set to the concept C?") and subsumption ("Is the set assigned to C necessarily a subset of the set assigned to D?"). A partial order representing all subsumption relations between the concept names contained in a TBox $\mathcal{T}$ is called the *subsumption hierarchy* of $\mathcal{T}$.

For an ABox $\mathcal{A}$, the individual names are assigned to individuals of the interpretation domain. Since DLs impose the *unique name assumption* on individual names, different individual names have to be interpreted by different individuals. The most important inference problems for ABoxes are the *consistency problem* and the *entailment problem*: $\mathcal{A}$ is consistent if it is possible to assign individuals of the interpretation domain to the individual names in such a way that the ABox constraints are respected, and $\mathcal{A}$ entails an assertion $\mathsf{C}(\mathsf{a})$ if every assignment that respects the constraints of $\mathcal{A}$ also satisfies $\mathsf{C}(\mathsf{a})$. In DLs allowing for negation of concept terms, the entailment problem can be reduced to the consistency problem: $\mathcal{A}$ entails $\mathsf{C}(\mathsf{a})$ iff $\mathcal{A} \cup \{\neg\mathsf{C}(\mathsf{a})\}$ is inconsistent; and the same holds for the satisfiability problem of concepts: C is

[2] Logics that allow for the internalisation of GCIs (see Section 2.3.6) do not fall into this category.

[3] Even together with role hierarchies, transitive roles are not as expressive as the transitive closure operator: the concept $\exists\mathsf{has\text{-}descendant}.\mathsf{Male} \sqcap \forall\mathsf{has\text{-}child}.\neg\mathsf{Male}$ and the RIA $\mathsf{has\text{-}child} \sqsubseteq \mathsf{has\text{-}descendant}$ together do not enforce a chain of $\mathsf{has\text{-}child}$ roles between the mother and the male descendant.

satisfiable iff $\{\mathsf{C}(\mathsf{a})\}$ is consistent for some individual name a. Conversely, in logics that allow for nominals (see Section 2.3.1), the ABox can be internalised in the TBox by replacing individual names with nominals. Since we will focus on TBoxes in the following, we will not go into details about ABox reasoning. Baader and Sattler (2001) present a prototypical tableau algorithm for testing ABox consistency in the basic DL $\mathcal{ALC}$, and Horrocks, Sattler, and Tobies (2000b) do the same for the more expressive logic $\mathcal{SHIQ}$.

In the following, we will describe in detail some examples for description logics of different expressivity.

2.3.4 Polynomial DLs

Since a DL allowing for all Boolean constructors ($\sqcap, \sqcup$ and $\neg$) is at least as expressive as propositional logic and therefore NP-hard, it is necessary to drop some of these constructors in order to obtain a DL with polynomial inference problems. One of the most basic DLs is called the *Existential Language ($\mathcal{EL}$)* and provides only existential quantification, conjunction, and the top concept (Baader et al., 1999).

Definition 2.1 ($\mathcal{EL}$ syntax). Let $\mathsf{N_C}$ be a set of concept names and $\mathsf{N_R}$ be a set of role names. The set of $\mathcal{EL}$ concept terms is defined as follows:

- every concept name $\mathsf{A} \in \mathsf{N_C}$ is a concept term;
- if C and D are concept terms, then $\top$ and $\mathsf{C} \sqcap \mathsf{D}$ are also concept terms;
- if C is a concept term and r is a role name, then $\exists \mathsf{r}.\mathsf{C}$ is also a concept term. $\diamond$

In the following, we will sometimes simply use the word "concept" for concept terms. The letters A, B will be used for concept names, whereas C, D etc. denote arbitrary concepts.

Definition 2.2 ($\mathcal{EL}$ TBox). For an $\mathcal{EL}$ concept name A and an $\mathcal{EL}$ concept term C, the expression $\mathsf{A} \doteq \mathsf{C}$ is called an $\mathcal{EL}$ *concept definition*. For two $\mathcal{EL}$ concept terms C and D, the expression $\mathsf{C} \sqsubseteq \mathsf{D}$ is called an $\mathcal{EL}$ *general concept inclusion axiom (GCI)*.

An *acyclic $\mathcal{EL}$ TBox* is a finite set of concept definitions such that every concept name occurs at most once as a left-hand side, and there is no cyclic dependency between the definitions, i. e. there is no sequence of concept definitions $\mathsf{A}_1 \doteq \mathsf{C}_1, \ldots, \mathsf{A}_n \doteq \mathsf{C}_n$ such that C_i contains A_{i+1} for $1 \leq i < n$ and C_n contains A_1.

A *general $\mathcal{EL}$ TBox* is an acyclic TBox extended with a finite set of GCIs.[4]

A concept name is called *defined* if it occurs on the left-hand side of a concept definition, and *primitive* otherwise. $\diamond$

Acyclicity of a TBox ensures that a concept name is not (directly or indirectly) defined by itself, and that concept definitions are not "abused" to define equivalence between complex concepts, e. g. as in $\{\mathsf{A} \doteq \exists \mathsf{r}.\mathsf{C}; \mathsf{A} \doteq \exists \mathsf{s}.\mathsf{D}\}$.

[4] This definition of general TBoxes is slightly non-standard since we do not allow for cyclic definitions. However, a general TBox with cyclic definitions can easily be transformed into an equivalent one without cyclic definitions by replacing concept definitions with GCIs as described in Section 2.3.1.

Definition 2.3 ($\mathcal{EL}$ Semantics and inference problems). For a set of concept names $\mathsf{N_C}$ and a set of role names $\mathsf{N_R}$, an *interpretation* $\mathcal{I}$ is a pair $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a set of individuals and $\cdot^\mathcal{I}$ is an interpretation function that maps concepts to unary relations and roles to binary relations over $\Delta^\mathcal{I}$. The interpretation function $\cdot^\mathcal{I}$ is extended to concept terms as follows:

- $\top^\mathcal{I} = \Delta^\mathcal{I}$;
- $(\mathsf{C} \sqcap \mathsf{D})^\mathcal{I} = \mathsf{C}^\mathcal{I} \cap \mathsf{D}^\mathcal{I}$;
- $(\exists \mathsf{r}.\mathsf{C})^\mathcal{I} = \{d \in \Delta^\mathcal{I} \mid$ there is an $e \in \Delta^\mathcal{I}$ with $(d, e) \in \mathsf{r}^\mathcal{I}$ and $e \in \mathsf{C}^\mathcal{I}\}$.

An interpretation $\mathcal{I}$ is called a *model* for a concept C if $\mathsf{C}^\mathcal{I} \neq \emptyset$. It is called a *model for a TBox* $\mathcal{T}$ if, for every definition $\mathsf{A} \doteq \mathsf{D} \in \mathcal{T}$ and for every GCI $\mathsf{E} \sqsubseteq \mathsf{F}$, it holds that $\mathsf{A}^\mathcal{I} = \mathsf{D}^\mathcal{I}$ and $\mathsf{E}^\mathcal{I} \subseteq \mathsf{F}^\mathcal{I}$. A model of C *with respect to* $\mathcal{T}$ is a model of C and $\mathcal{T}$. A concept C is called *satisfiable (with respect to* $\mathcal{T}$*)* if there is a model of C (and $\mathcal{T}$). Similarly, we say that a concept C is *subsumed* by a concept D, written as $\mathsf{C} \sqsubseteq \mathsf{D}$ (w. r. t. $\mathcal{T}$) if, in every interpretation (every model for $\mathcal{T}$), $\mathsf{C}^\mathcal{I} \subseteq \mathsf{D}^\mathcal{I}$ holds. ◇

Since $\mathcal{EL}$ does not allow for negation, the satisfiability problem is not interesting (every concept term is satisfiable). However, subsumption is not trivial, and it was shown by Baader, Küsters, and Molitor (1998) that the subsumption problem for $\mathcal{EL}$ concept terms is decidable in polynomial time. This complexity result also holds for a rather expressive extension of $\mathcal{EL}$, named $\mathcal{EL}^{++}$, which additionally allows for GCIs, role inclusion axioms, nominals, and concrete domains (Baader et al., 2005). In Chapter 5, we will see that in $\mathcal{ELU}_f$, $\mathcal{EL}$ extended with disjunction and features, satisfiability w. r. t. GCIs is ExpTime-hard. Recently, this result has been sharpened by showing that satisfiability in $\mathcal{EL}$ augmented with *either* features *or* disjunction becomes ExpTime-hard (Baader et al., 2005).

DLs that allow only for value restriction and not for existential restriction have also been examined (Nebel, 1990; Donini, Lenzerini, Nardi, and Nutt, 1991; Baader, 1996; Kazakov and de Nivelle, 2003). Since these logics are not used in the remainder of this thesis, we will not describe them in detail.

2.3.5 PSpace DLs

If we add negation of concept terms ($\neg$) to $\mathcal{EL}$, we can also express the constructors $\sqcup$, $\forall$, and $\bot$ due to their duality to the $\mathcal{EL}$ ones. The language that is obtained by adding all of these constructors to $\mathcal{EL}$ is called the *Attributive Language with Complements,* $\mathcal{ALC}$ (Schmidt-Schauß and Smolka, 1991).

Definition 2.4 ($\mathcal{ALC}$ syntax and semantics). The set of $\mathcal{ALC}$ concepts is defined as in Definition 2.1, with the following additions:

- if C and D are concept terms, then $\bot$, $\neg\mathsf{C}$, and $\mathsf{C} \sqcup \mathsf{D}$ are also concept terms;
- if C is a concept term and r is a role name, then $\forall \mathsf{r}.\mathsf{C}$ is also a concept term.

The semantics for the additional constructors is defined as follows:

- $\bot^{\mathcal{I}} = \emptyset$;
- $(\neg \mathsf{C})^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \mathsf{C}^{\mathcal{I}}$;
- $(\mathsf{C} \sqcup \mathsf{D})^{\mathcal{I}} = \mathsf{C}^{\mathcal{I}} \cup \mathsf{D}^{\mathcal{I}}$;
- $(\forall \mathsf{r}.\mathsf{C})^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \text{ for all } e \in \Delta^{\mathcal{I}} \text{ with } (d,e) \in \mathsf{r}^{\mathcal{I}}, e \in \mathsf{C}^{\mathcal{I}} \text{ holds}\}$. ◇

Subsumption and satisfiability are defined as for $\mathcal{EL}$. Since $\mathcal{ALC}$ allows for conjunction and negation, these inference problems can be mutually reduced to each other: C is satisfiable if it is not subsumed by $\bot$, and C is subsumed by D if the concept $\mathsf{C} \sqcap \neg \mathsf{D}$ is unsatisfiable. In the following, we will therefore only consider the satisfiability problem. $\mathcal{ALC}$ concept satisfiability is PSPACE-complete: PSPACE-hardness can be shown via a reduction of the validity problem for quantified boolean formulas, QBF (Schmidt-Schauß and Smolka, 1991). A PSPACE tableau algorithm deciding $\mathcal{ALC}$ concept satisfiability is sketched in Section 3.2.1. In Chapter 6, we show how the techniques used to achieve this result can be transferred to automata algorithms.

As mentioned in the beginning of this section, Schild (1991) discovered that $\mathcal{ALC}$ is a notational variant of the multi-modal logic $\mathbf{K}_m$ (see e. g. Blackburn et al., 2001) and that, consequently, the $\mathcal{ALC}$ tableau algorithm is a reinvention of the the known tableau algorithm for satisfiability in $\mathbf{K}_m$. Similarly, if we extend $\mathcal{ALC}$ with transitive roles, we obtain a logic that, if all roles are transitive, corresponds to the modal logic $\mathbf{S4}_m$. Due to this analogy, we will denote this DL by $\mathcal{S}$ in the following. PSPACE-completeness of $\mathcal{S}$ can be shown in a similar way as for $\mathbf{S4}_m$. A further extension of $\mathcal{S}$ that remains within the same complexity class is $\mathcal{SI}$, $\mathcal{S}$ extended with with *inverse roles*: concept satisfiability for $\mathcal{SI}$ is PSPACE-complete (Horrocks, Sattler, and Tobies, 1999), and we will see in Section 6.2 that this also holds in the presence of acyclic TBoxes. The reason is that acyclic TBoxes only allow for more concise abbreviations, and they can be handled in an algorithmically more efficient way than general TBoxes.

2.3.6 EXPTIME DLs

The satisfiability problem for $\mathcal{ALC}$ w. r. t. general TBoxes is EXPTIME-complete, which can be shown by a reduction of the word problem for linear space bounded alternating Turing machines (McAllester, Givan, Witty, and Kozen, 1996), similar to the method of Fischer and Ladner (1979) for **PDL**. In DLs that provide transitive roles, role hierarchies, negation and disjunction, it is possible to *internalise* general TBoxes (Horrocks and Sattler, 1999; Horrocks, Sattler, and Tobies, 2000a), in a way that is similar to the one used by Baader (1991). Here, internalising a TBox $\mathcal{T}$ means translating $\mathcal{T}$ into a concept $\mathsf{C}_{\mathcal{T}}$ of polynomial size such that a concept D is satisfiable w. r. t. $\mathcal{T}$ iff $\mathsf{D} \sqcap \mathsf{C}_{\mathcal{T}}$ is satisfiable (w. r. t. the empty TBox). Before describing this method in detail, we define the corresponding DL.

Definition 2.5 ($\mathcal{SH}$ syntax and semantics). Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be as in Definition 2.1, with $\{\mathsf{C}, \mathsf{D}\} \subseteq \mathsf{N_C}$ and $\{\mathsf{r}, \mathsf{s}\} \subseteq \mathsf{N_R}$. Additionally, let $\mathsf{N_{R^+}}$, the set of *transitive role names*, be a subset of $\mathsf{N_R}$. The set of $\mathcal{SH}$ *concepts* is defined as for $\mathcal{ALC}$.

A *role inclusion axiom (RIA)* has the form $\mathsf{r} \sqsubseteq \mathsf{s}$. An *RBox* is a finite set of role inclusion axioms. The semantics of transitive roles and RBoxes $\mathcal{B}$ is as follows:

Constructor	Syntax	Semantics
top	$\top$	$\Delta^{\mathcal{I}}$
bottom	$\bot$	$\emptyset$
conjunction	$\mathsf{C} \sqcap \mathsf{D}$	$\mathsf{C}^{\mathcal{I}} \cap \mathsf{D}^{\mathcal{I}}$
disjunction	$\mathsf{C} \sqcup \mathsf{D}$	$\mathsf{C}^{\mathcal{I}} \cup \mathsf{D}^{\mathcal{I}}$
negation	$\neg\mathsf{C}$	$\Delta^{\mathcal{I}} \setminus \mathsf{C}^{\mathcal{I}}$
existential r.	$\exists\mathsf{r}.\mathsf{C}$	$\{d \in \Delta^{\mathcal{I}} \mid$ there is an e with $(d,e) \in \mathsf{r}^{\mathcal{I}}$ and $e \in \mathsf{C}^{\mathcal{I}}\}$
value r.	$\forall\mathsf{r}.\mathsf{C}$	$\{d \in \Delta^{\mathcal{I}} \mid$ for every e with $(d,e) \in \mathsf{r}^{\mathcal{I}}, e \in \mathsf{C}^{\mathcal{I}}$ holds$\}$
qualif. at-most	$(\leqslant n\ \mathsf{r}\ \mathsf{C})$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{e \mid (d,e) \in \mathsf{r}^{\mathcal{I}}$ and $e \in \mathsf{C}^{\mathcal{I}}\} \leq n\}$
qualif. at-least	$(\geqslant n\ \mathsf{r}\ \mathsf{C})$	$\{d \in \Delta^{\mathcal{I}} \mid \#\{e \mid (d,e) \in \mathsf{r}^{\mathcal{I}}$ and $e \in \mathsf{C}^{\mathcal{I}}\} \geq n\}$
inverse roles	r^-	$\{(d,e) \mid (e,d) \in \mathsf{r}^{\mathcal{I}}\}$
transitive roles	$\mathsf{t} \in \mathsf{N}_{\mathsf{R}^+}$	$\mathsf{t}^{\mathcal{I}} = (\mathsf{t}^{\mathcal{I}})^+$
features	$\mathsf{f} \in \mathsf{N}_{\mathsf{F}}$	for every $d \in \Delta^{\mathcal{I}}, \#\{e \mid (d,e) \in \mathsf{f}^{\mathcal{I}}\} \leq 1$
nominals	$\mathsf{C} \in \mathsf{N}_{\mathsf{O}}$	$\#\mathsf{C}^{\mathcal{I}} = 1$
concept def.	$\mathsf{A} \doteq \mathsf{C}$	$\mathsf{A}^{\mathcal{I}} = \mathsf{C}^{\mathcal{I}}$
GCI	$\mathsf{C} \sqsubseteq \mathsf{D}$	$\mathsf{C}^{\mathcal{I}} \subseteq \mathsf{D}^{\mathcal{I}}$
RIA	$\mathsf{r} \sqsubseteq \mathsf{s}$	$\mathsf{r}^{\mathcal{I}} \subseteq \mathsf{s}^{\mathcal{I}}$

Figure 2.1: DL constructors

- for every $\mathsf{t} \in \mathsf{N}_{\mathsf{R}^+}$, it holds that $\mathsf{t}^{\mathcal{I}} = (\mathsf{t}^{\mathcal{I}})^+$, i. e. for individuals a, b, and c, $\{(a,b),(b,c)\} \subseteq \mathsf{t}^{\mathcal{I}}$ implies $(a,c) \in \mathsf{t}^{\mathcal{I}}$.
- for every $\mathsf{r} \sqsubseteq \mathsf{s} \in \mathcal{B}$, it holds that $\mathsf{r}^{\mathcal{I}} \subseteq \mathsf{s}^{\mathcal{I}}$. $\diamond$

In $\mathcal{SH}$, it is possible to internalise TBoxes as follows: one defines a new role u as a transitive role that comprises all other roles, which implies that every individual j that is reachable from another individual i is reachable from i via u. Moreover, since $\mathcal{SH}$ has the *connected model property*, i. e. every satisfiable concept has a connected model,[5] every relevant individual in a model for the concept C is reachable from the individual that is a witness for C. Thus, if one defines, for a TBox $\mathcal{T} = \{\mathsf{D}_1 \sqsubseteq \mathsf{E}_1, \ldots, \mathsf{D}_n \sqsubseteq \mathsf{E}_n\}$ and an RBox $\mathcal{B}$, the concept $\mathsf{C}_{\mathcal{T}} := (\neg\mathsf{D}_1 \sqcup \mathsf{E}_1) \sqcap \ldots \sqcap (\neg\mathsf{D}_n \sqcup \mathsf{E}_n)$ and the RBox $\mathcal{B}' := \mathcal{B} \cup \{\mathsf{r} \sqsubseteq \mathsf{u} \mid \mathsf{r} \in \mathsf{N}_{\mathsf{R}}\}$, then the size of $\mathsf{C}_{\mathcal{T}}$ is linear in the size of $\mathcal{T}$, and a concept C is satisfiable w. r. t. $\mathcal{T}$ and $\mathcal{B}$ iff the concept $\mathsf{C} \sqcap \mathsf{C}_{\mathcal{T}} \sqcap \forall\mathsf{u}.\mathsf{C}_{\mathcal{T}}$ is satisfiable w. r. t. $\mathcal{B}'$ (and the empty TBox).

$\mathcal{SH}$ allows for several ExpTime-complete extensions. For example, adding qualifying number restrictions (QNR) and nominals gives $\mathcal{SHOQ}$ (Horrocks and Sattler, 2001), and adding QNR and inverse roles leads to $\mathcal{SHIQ}$ (Horrocks et al., 2000a), which is the basis for most current implementations (see Section 2.3.7). In Section 7.4 we show that also $\mathcal{SHIO}$, $\mathcal{SH}$ augmented with nominals and inverse roles, remains in ExpTime. However, if all three constructors are combined, the resulting DL $\mathcal{SHOIQ}$ becomes NExpTime-hard, which can be shown using a reduction from a bounded

[5] This holds since **PDL** has the connected model property (see e. g. De Giacomo, 1996), $\mathcal{S}$ is a sublanguage of **PDL**, and adding role hierarchies clearly does not allow for the introduction of non-connected individuals.

DL	top	bottom	conjunction	disjunction	negation	existential restriction	value restriction	inverse roles	transitive roles	role hierarchies	QNR (at-least/at-most)	features	nominals
$\mathcal{EL}$	✓		✓			✓							
$\mathcal{ELU}_f$	✓	✓	✓	✓		✓						✓	
$\mathcal{FLEUI}_f$	✓	✓	✓	✓		✓	✓	✓				✓	
$\mathcal{ALC}$	✓	✓	✓	✓	✓	✓	✓						
$\mathcal{ALCIO}$	✓	✓	✓	✓	✓	✓	✓	✓					✓
$\mathcal{SI}$	✓	✓	✓	✓	✓	✓	✓	✓	✓				
$\mathcal{SH}$	✓	✓	✓	✓	✓	✓	✓		✓	✓			
$\mathcal{SHIQ}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(✓)	
$\mathcal{SHOQ}$	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	(✓)	✓
$\mathcal{SHIO}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
$\mathcal{SHOIQ}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(✓)	✓

Figure 2.2: Description logics and corresponding constructors

version of the domino problem (Tobies, 2000). An upper NExpTime bound follows from the containedness of $\mathcal{SHOIQ}$ in C^2, the two-variable fragment of predicate logic with counting (Pacholski, Szwast, and Tendera, 1997).

Figure 2.1 gives an overview of the constructors described in this section and their semantics; there $\#S$ denotes the cardinality of a set S. In Figure 2.2, we summarise which languages allow for which constructors. The logics providing QNR (i. e. $\mathcal{SH}(\mathcal{O})(\mathcal{I})\mathcal{Q}$) do not explicitly provide features, but it is possible to express functionality of a role f with QNR by replacing $\exists \mathsf{f}.\mathsf{C}$ with $(\geqslant 1\ \mathsf{f}\ \mathsf{C}) \sqcap (\leqslant 1\ \mathsf{f}\ \top)$ or, in the presence of general TBoxes, by adding the GCI $\top \sqsubseteq (\leqslant 1\ \mathsf{f}\ \top)$. Although some of the listed logics appear only in later chapters, they are included in the table for easier reference. Figure 2.3 shows the known complexity results for these DLs. For $\mathcal{EL}$, this refers to the complexity of the subsumption problem (since satisfiability is trivial); for all other DLs, the complexity result holds for both subsumption and satisfiability. Known results are displayed in grey.

2.3.7 DL Systems

In order to use a DL in real-life applications, it should be possible to perform the inference tasks in acceptable time, where "acceptable" usually means "polynomial", and NP-hard problems are already considered to be "intractable". However, Nebel (1990) early discovered that even rather inexpressive DLs are intractable, which raised the question if DLs (at that time called KL-One-based languages) could be used in

DL	concept	acyclic TBox	general TBox
$\mathcal{EL}$	P	P	P
$\mathcal{ELU}_f$			ExpTime-complete
$\mathcal{ALC}$	PSpace-complete	PSpace-complete	ExpTime-complete
$\mathcal{SI}$	PSpace-complete	PSpace-complete	ExpTime-complete
$\mathcal{SH}$	ExpTime-complete		
$\mathcal{SHIQ}$	ExpTime-complete		
$\mathcal{SHOQ}$	ExpTime-complete		
$\mathcal{SHIO}$	ExpTime-complete		
$\mathcal{SHOIQ}$	NExpTime-complete		

Figure 2.3: Complexity of the satisfiability (subsumption) problem

practice at all. Some early systems circumvented this obstacle by using only a very inexpressive DL with a modified semantics and thus achieving tractability, as in the case of Classic (Borgida, Brachman, McGuinness, and Alperin Resnick, 1989; Patel-Schneider, McGuiness, Brachman, Alperin Resnick, and Borgida, 1991), or by using incomplete reasoning algorithms, as in the case of Loom (MacGregor and Bates, 1987)[6]. However, the Kris system (Baader and Hollunder, 1991a,b) showed that it is possible to achieve acceptable performance for an intractable DL (more precisely $\mathcal{ALCNF}$, i. e. $\mathcal{ALC}$ extended with non-qualifying number restrictions and features) using a sound and complete tableau algorithm for the satisfiability test. Kris optimises the computation of the subsumption hierarchy by exploiting information that is explicitly contained in the TBox (so-called *told subsumers*), and by carefully choosing the order in which subsumption tests are performed in order to avoid redundancy.

The FaCT system (Horrocks, 1997, 1998a) continued this work by showing that even $\mathcal{SHIQ}$ can be handled efficiently. The most powerful optimisations implemented in FaCT are *backjumping*, which avoids redundant tests during backtracking, and *semantic branching*, which makes found contradictions explicit and thus aims at reducing the search space (for a detailed description of these optimisations, see Section 3.2.1). Subsequently, the $\mathcal{SHIQ}$ tableau algorithm was also implemented in the systems Racer (Haarslev and Möller, 2001b), RacerPro (Haarslev, Möller, and Wessel, 2005a), and Pellet (Sirin and Parsia, 2004) which, like FaCT, have exhibited good performance on "realistic" knowledge bases (see e. g. Haarslev and Möller, 2001a; Haarslev, Möller, and Wessel, 2005b; Sirin, Parsia, Cuenca Grau, Kalyanpur, and Katz, 2007). In their most recent versions, FaCT++ (Tsarkov and Horrocks, 2006), which is the successor of FaCT, and Pellet (Sirin et al., 2007) have been extended to the DL $\mathcal{SHOIQ}$, using the tableau algorithm developed by Horrocks and Sattler (2005).

Due to the analogies between description and modal logics, satisfiability testers for modal logics can also be used for testing satisfiability of DL expressions. Two ML systems that do not follow the tableau paradigm are Kk (Voronkov, 1999, 2001) and

[6] At that time, the polynomial DLs of the $\mathcal{EL}$ family described in Section 2.3.4 were not examined in detail because value restriction was regarded as indispensable.

the BDD-based procedure introduced by Pan et al. (2006) (see Section 1.3). In spite of the fact that they are based on different ideas (KK implements the *inverse method*, which has no obvious relations with binary decision diagrams), both systems can be seen as optimised implementations of automata algorithms. The DPLL-based reasoner KSAT developed by Giunchiglia and Sebastiani (1996) exhibited good performance on the test concepts generated by the developers, but the significance of these results is questionable (Hustadt and Schmidt, 1997). However, some of the criticism of tableau algorithms formulated by Giunchiglia and Sebastiani (1996) has led to improvements of tableau algorithms, most notably by the introduction of *semantic branching* (Horrocks and Patel-Schneider, 1999). As mentioned in Section 1.3, the MSPASS system (Hustadt and Schmidt, 2000) uses resolution to decide satisfiability for different modal logics.

The recently developed KAON2 system (Hustadt et al., 2004) follows a different approach: a $\mathcal{SHIQ}$ TBox is transformed into a disjunctive datalog program, such that ABox reasoning can be performed using techniques developed for deductive databases. Experiments have shown that this method is superior to tableau-based systems for small TBoxes and large ABoxes, but inferior if the TBox is large (Motik and Sattler, 2006). It is an interesting open question whether this approach can be optimised for large TBoxes and whether it can be extended to NEXPTIME-hard logics like $\mathcal{SHOIQ}$. Similarly, first results with the hypertableau calculus on real-life knowledge bases are promising (Motik et al., 2007), but it is an open question how this algorithm will perform on inputs that cannot be translated into horn clauses.

Chapter 3

Tableau Algorithms

Tableau algorithms (TAs) are a class of algorithms that are employed for testing satisfiability in various logics, e. g. "classical" propositional and predicate logic, modal and temporal logics, intuitioninstic logic, and non-monotonic logics (D'Agostino, Gabbay, Hähnle, and Posegga, 1999).

The term *tableau algorithm* originates from a method for disproving validity of formulas, called *semantic tableaus*, introduced by Beth (1955). The name *tableaus* results from the fact that such an algorithm sets up a special kind of table for the variables and their truth assignments. These tableaus are called *semantic* because, in order to test the validity of a formula φ, they attempt to find a counter-model for φ and thus operate on a semantic level instead of the syntactic *sequent calculus* by Gentzen (1935).

In this chapter, we will first describe tableau algorithms for propositional logic, and then show how this mechanism is extended to deal with description logics. As examples, we define TAs deciding concept satisfiability for $\mathcal{ALC}$ with and without general TBoxes.

3.1 Tableaus for Propositional Logic

In order to test the validity of a propositional formula φ, a tableau algorithm exhaustively searches for a valuation of the propositional variables in φ that evaluates φ to *false*. For this purpose, a table with two columns is set up, one for sub-formulas of φ that have to evaluate to *true*, and one for formulas that have to evaluate to *false*. Starting with φ in the *false* column, every complex formula is split into simpler formulas according to *tableau rules*; e. g. if a formula $A \wedge B$ appears in a column, both A and B are added to that column, and for $\neg C$, C is added to the other column. For a disjunction $A \vee B$, both alternatives have to be tested, and thus every column is *split*, with A (and every formula resulting from further rule applications to A) being added to the first sub-column, and B to the second.

A *tableau rule* is characterised by three properties: a *precondition*, which establishes when to apply the rule (e.g. "if $A \wedge B$ appears in a column"); a *postcondition*, which determines what to do ("add A and B to the column"); and an *applicability*

condition, which prevents a rule from being applied infinitely often ("if A and B are not both present in the column"). For propositional logic, the applicability conditions are obvious, but in the presence of quantifiers, things become more involved (see Section 3.2.1).

The algorithm terminates if, for every sub-column, a propositional variable A appears on both the *true* and the *false* side, which implies a contradiction (the sub-column is then called *closed*), or if in one of the sub-columns, all applicable rules have been applied on both sides without causing a contradiction (i. e. there remains an *open* column). In the former case, this means that φ is valid because no valuation can be found that evaluates φ to *false*; in the latter case, we have found a counter-model, and thus φ is invalid.

During the run of the tableau algorithm, two kinds of non-determinism occur: if several rules are applicable, the choice which rule to apply first is *don't-care*-non-deterministic, i. e. either decision will lead to the same result, provided that every applicable rule is applied eventually, a property which is referred to as *fairness*. On the other hand, the decision which disjunct of a disjunction to test is *don't-know*-non-deterministic, i. e. one decision might lead to a counter-model, while the other one might not. For this reason, the table columns have to be split for disjunctions, but not for different sequences of rule application.

The image of this table recursively split into sub-tables created the impression of a *tableau*. In present-day computer science, this might remind the viewer of a binary tree rather than a tableau, with the first branch standing for the division of true and false formulas, and every subsequent one standing for a disjunction. In fact, an advanced version of semantic tableaus developed by Hintikka (1955) uses a tree structure instead of a table. More importantly, Hintikka has simplified the mechanism of tableaus by only allowing for formulas in *negation normal form* (NNF), i. e. only atomic formulas can be negated, and by introducing *downward saturated sets*, i. e. sets which are propositionally expanded and contradiction-free. Both techniques are still standard for DLs: the TAs in this chapter and the remainder of this thesis are defined for concepts in NNF, and the algorithm in 4.3.1 relies on the fact that every downward saturated set (there called *Hintikka set*) is satisfiable, which was proved by Hintikka (1955).

Smullyan (1968) avoided having to deal with two trees in parallel (one for *true* and one for *false* formulas) by explicitly labelling every appearing formula with "T" or "F" *(signed tableau)* or by using negation ($\neg$) for formulas that have to evaluate to *false* *(unsigned tableau)*. He also coined the term *analytic tableau* to emphasise that, in the tableau for φ, only subformulas of φ can appear (Beth (1959) had already shown the *subformula principle*, i. e. only subformulas of φ are required to prove φ).

Fitting (1999) considers three features to be essential for all kinds of tableau algorithms: they are *refutation* procedures, which try to find a counter-model for the input; they operate by applying *rules* that break the input formula down syntactically; and they detect a contradiction in the generated formula set using *closing cases*. These properties, which already characterised the first tableau systems by Beth, are still present in current tableau algorithms for description logics (see e. g. Baader and Sattler, 2001; Baader and Nutt, 2003).

3.2 Tableaus for Description Logics

In the context of modal and description logics, one usually does not describe the current step in the generation of a model by a set of concepts (although this is possible—see the ABox consistency algorithm by Baader and Sattler (2001)), but rather by a data structure called *completion tree*, which suggests itself since most DLs have the *tree model property* (i. e. every satisfiable input has a model in the shape of a tree), or at least some kind of relaxed tree model property, e. g. $\mathcal{SHOQ}$ (Horrocks and Sattler, 2001) has a *forest* model property, i. e. every satisfiable input has a model that consists of a *set* of trees.

Such a completion tree consists of a set V of nodes and a set $E \subseteq V \times V$ of edges, together with labelling functions n and ℓ, which assign to every node a set of concept terms and to every edge a set of roles or, in simple cases, just a single role. Intuitively, a node stands for an individual and the node label for the concepts that the individual satisfies, whereas an edge stands for a relation of two individuals by a role. The root node of the completion tree is labelled with the input concept, and a successor w of a node v stands for an individual that satisfies an existential restriction of v.

Using a tree as data structure for models has several advantages. From a theoretical point of view, this shape is often helpful to achieve termination: for logics that do not have the finite model property, it is necessary to avoid the generation of an unbounded number of individuals. Since the subformula principle also holds for most DLs, there can be only a finite number of different types for individuals, and it is unnecessary to create two individuals with the same type (we say that one individual *blocks* the generation of the other one). In order to show soundness of this approach, it is necessary to ensure that two individuals do not mutually block each other, and the partial order induced by the structure of the tree can be used for that purpose (for details, see Section 3.2.2). From a practical point of view, the tree structure improves efficiency because the applicability of most rules depends only on the concepts present in one node and possibly its immediate predecessors or successors. Therefore, if a tree structure is used, the time required to test for the presence of a certain concept in the label of a certain individual will be independent on the total number of individuals, whereas this time would increase if all concepts were kept in one set.

Thus, the data structure of a DL tableau algorithm is a tree of trees, where the "outer" one is a binary tree that has a branching point for every non-deterministic decision (as in propositional tableaus) and contains at every node an "inner" tree that is a completion tree.

We will now look at two DL tableau algorithms in detail. The main features mentioned in Section 3.1 for propositional tableaus also characterise DL tableaus: they are *refutation* procedures since the satisfiability test is usually performed for a concept $\mathsf{C} \sqcap \neg\mathsf{D}$ in order to refute the subsumption $\mathsf{C} \sqsubseteq \mathsf{D}$; they use a set of *rules* in order to break the input down syntactically; and they detect unsatisfiability using a set of closing cases which, in the DL context, are called *clash-triggers*.

```
create a node v_0 with n(v_0) = C
while a rule is applicable to a node v do
  apply a rule to v
  if the completion tree is closed then
    return "unsatisfiable"
  end if
end while
return "satisfiable"
```

Figure 3.1: Tableau algorithm for $\mathcal{ALC}$ concept satisfiability

3.2.1 A Tableau Algorithm for $\mathcal{ALC}$ Concept Satisfiability

As a first example for a DL tableau algorithm, we will present an algorithm for $\mathcal{ALC}$ concept satisfiability. For the sake of simplicity, we will assume that all concept terms are in *negation normal form (NNF)*, i. e. negation only appears directly before concept names. All $\mathcal{ALC}$ concepts can be transformed into NNF in linear time by using DeMorgan's laws, the duality of the quantifiers and by removing double negation. We will denote the NNF of a concept C by $\mathsf{nnf}(\mathsf{C})$ and $\mathsf{nnf}(\neg\mathsf{C})$ by $\dot{\neg}\mathsf{C}$.

Let $\mathcal{ALC}$ be as in Definition 2.4. Then the satisfiability of an $\mathcal{ALC}$ concept C can be tested by the algorithm in Figure 3.1 using the rules given in Figure 3.2. These rules clearly show the three properties of tableau rules mentioned in Section 3.1: the precondition is given in the first line, the applicability condition in the second one and the postcondition in the third one. Note that the $\sqcup$-rule is non-deterministic since it can add either of the two disjuncts to the corresponding node label. Regarding non-deterministic rules, we assume that the output of the algorithm is "satisfiable" if there *exists* a sequence of rule applications that leads to the output "satisfiable".

Definition 3.1 (Subconcept, completion tree, tableau). Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be as in Definition 2.1, and let C be an $\mathcal{ALC}$ concept term in NNF. The set of *subconcepts of* C, $\mathit{sub}(\mathsf{C})$, is the minimal set S which contains C and has the following properties:

- if S contains $\neg\mathsf{A}$ for a concept name A, then $\mathsf{A} \in S$;
- if S contains $\mathsf{D} \sqcup \mathsf{E}$ or $\mathsf{D} \sqcap \mathsf{E}$, then $\{\mathsf{D}, \mathsf{E}\} \subseteq S$;
- if S contains $\exists\mathsf{r}.\mathsf{D}$ or $\forall\mathsf{r}.\mathsf{D}$, then $\mathsf{D} \in S$.

The set $\mathit{rol}(\mathsf{C})$ is the set of role names appearing in C.

A *completion tree* for C is a tuple $T = (V, E, n, \ell)$, where (V, E) is a tree and $n : V \to \mathit{sub}(\mathsf{C})$ and $\ell : E \to \mathit{rol}(\mathsf{C})$ are node and edge labelling functions, respectively. A completion tree is *closed* if there exists a node $v \in V$ with $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq n(v)$ for some $\mathsf{A} \in \mathsf{N_C}$. Otherwise T is called *open*. Moreover, T is called *saturated*[1] if no rule is applicable to a node of T. An open and saturated completion tree is called *tableau*.

If, for two nodes v, w, it holds that $(v, w) \in E$ and $\ell(v, w) = \mathsf{r}$, then w is called an r-*successor* of v. ◇

[1] Usually, this property is called "completeness", however, in order to avoid confusion with the notion of completeness of the decision procedure, we will use the term "saturated".

$\sqcap$**-rule** **If** for a node v, $n(v)$ contains a concept $\mathsf{C} \sqcap \mathsf{D}$
and $\{\mathsf{C}, \mathsf{D}\} \not\subseteq n(v)$
then set $n(v) := n(v) \cup \{\mathsf{C}, \mathsf{D}\}$.

$\sqcup$**-rule** **If** for a node v, $n(v)$ contains a concept $\mathsf{C} \sqcup \mathsf{D}$
and $n(v) \cap \{\mathsf{C}, \mathsf{D}\} = \emptyset$
then choose E from $\{\mathsf{C}, \mathsf{D}\}$ and set $n(v) := n(v) \cup \{\mathsf{E}\}$.

$\forall$**-rule** **If** for a node v, $n(v)$ contains a concept $\forall \mathsf{r}.\mathsf{C}$
and there is an r-successor w of v with $\mathsf{C} \notin n(w)$,
then set $n(w) := n(w) \cup \{\mathsf{C}\}$.

$\exists$**-rule** **If** for a node v, $n(v)$ contains a concept $\exists \mathsf{r}.\mathsf{C}$
and there is no r-successor w of v with $\mathsf{C} \in n(w)$,
then create a new r-successor w of v with $n(w) = \{\mathsf{C}\}$.

Figure 3.2: Tableau rules for $\mathcal{ALC}$

In the context of DLs, the presence of A and $\neg\mathsf{A}$ in a node label is also called a *clash* because it indicates a conflict between two conditions that the completion tree has to satisfy.

Although the TA for $\mathcal{ALC}$ concept satisfiability is well-known, we will formally prove its correctness because the details of the proofs are part of our comparison of tableau and automata algorithm.

Theorem 3.2. The tableau algorithm described in Figures 3.1 and 3.2 effectively decides $\mathcal{ALC}$ concept satisfiability.

Proof. We have to show soundness, completeness, and termination.

Soundness. Let (V, E, n, ℓ) be a tableau for C. We generate a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ as follows:

- $\Delta^{\mathcal{I}} := V$;
- For a concept name D, $\mathsf{D}^{\mathcal{I}} := \{v \in V \mid \mathsf{D} \in n(v)\}$;
- For a role name r, $\mathsf{r}^{\mathcal{I}} := \{(v, w) \mid (v, w) \in E \text{ and } \ell(v, w) = \mathsf{r}\}$

It follows by induction over the structure of concept terms that the individual v satisfies all concepts in the label of the node v, i. e. that $\mathsf{D} \in n(v)$ implies $v \in \mathsf{D}^{\mathcal{I}}$:

- for a concept name D, this follows from the definition of $\mathsf{D}^{\mathcal{I}}$;
- for a negated concept name $\neg\mathsf{D}$, since the tableau is open, $n(v)$ does not contain D, and therefore $v \notin \mathsf{D}^{\mathcal{I}}$ holds by the definition of $\mathsf{D}^{\mathcal{I}}$;
- for a conjunction [disjunction] $\mathsf{D} \sqcap [\sqcup]\mathsf{E}$, since the tableau is saturated, $n(v)$ contains both [one of] D and E, and by induction $v \in \mathsf{D}^{\mathcal{I}} \cap [\cup]\mathsf{E}^{\mathcal{I}}$ holds;

- for an existential restriction $\exists \mathsf{r}.\mathsf{D}$, since the tableau is saturated, there exists an r-successor w of v labelled with D. By construction, it also holds that $(v, w) \in \mathsf{r}^{\mathcal{I}}$ and, by induction, $w \in \mathsf{D}^{\mathcal{I}}$, which implies $v \in \exists \mathsf{r}.\mathsf{D}$.

- for a value restriction $\forall \mathsf{r}.\mathsf{D}$, let w be an individual such that $(v, w) \in \mathsf{r}^{\mathcal{I}}$. (If such an individual does not exist, there is nothing to show.) Then, by the construction of $\mathcal{I}$, w is an r-successor of v in the tableau, which implies that $n(w)$ contains D because the tableau is saturated, and by induction w belongs to $\mathsf{D}^{\mathcal{I}}$.

Completeness. Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be a model for C. Then $\mathcal{I}$ can be used to find a sequence of rule applications that the algorithm in Figure 3.1 can guess and thus construct a clash-free and saturated completion tree (V, E, n, ℓ). For this purpose, we will define a function $\varphi : V \to \Delta^{\mathcal{I}}$ such that

1. for a node $v \in V$, it holds that $n(v) \subseteq \{\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid \varphi(v) \in \mathsf{D}^{\mathcal{I}}\}$ and

2. for two nodes $v, w \in V$, if w is an r-successor of v, then $(v, w) \in \mathsf{r}^{\mathcal{I}}$ holds.

We start by mapping the root node of the completion tree to an element of $\mathsf{C}^{\mathcal{I}}$. Such an element exists because $\mathcal{I}$ is a model for C, and it obviously satisfies both conditions. We now show, for every rule application, how the model can guide the application of the corresponding rule in such a way that we can extend φ without violating either condition:

The $\sqcap$-rule: If $n(v)$ contains $\mathsf{D} \sqcap \mathsf{E}$, then $\varphi(v)$ belongs to $(\mathsf{D} \sqcap \mathsf{E})^{\mathcal{I}}$ by Condition 1 and therefore also to $(\mathsf{D}^{\mathcal{I}} \cap \mathsf{E}^{\mathcal{I}})$. Thus both conditions are still satisfied after the rule application.

The $\sqcup$-rule: Similarly, if $n(v)$ contains $\mathsf{D} \sqcup \mathsf{E}$, then $\varphi(v)$ belongs to $(\mathsf{D}^{\mathcal{I}} \cup \mathsf{E}^{\mathcal{I}})$, and thus the $\sqcup$-rule can choose a concept from $\{\mathsf{D}, \mathsf{E}\}$ in such a way that the conditions are preserved.

The $\exists$-rule: If $n(v)$ contains $\exists \mathsf{r}.\mathsf{D}$, then $\varphi(v)$ belongs to $(\exists \mathsf{r}.\mathsf{D})^{\mathcal{I}}$, i. e. there is an element $d \in \mathsf{D}^{\mathcal{I}}$ such that $(\varphi(v), d) \in \mathsf{r}^{\mathcal{I}}$. Thus we can apply the rule to add a new r-successor w of v with $n(w) = \{\mathsf{D}\}$. If we define $\varphi(w) = d$, then φ still satisfies both conditions.

The $\forall$-rule: If $n(v)$ contains $\forall \mathsf{r}.\mathsf{D}$ and there is an r-successor w of v, then it holds that $\varphi(v) \in (\forall \mathsf{r}.\mathsf{D})^{\mathcal{I}}$ (by Condition 1) and $(v, w) \in \mathsf{r}^{\mathcal{I}}$ (by Condition 2). Together, these observations imply $w \in \mathsf{D}^{\mathcal{I}}$, thus we can add D to $n(w)$ without violating the conditions.

Finally, the completion tree resulting from this sequence of rule applications does not contain a clash since all node labels are based on the sets of concepts that the corresponding individuals satisfy, and these sets are clash-free because $\mathcal{I}$ is a model.

Termination. Each node label is a subset of $sub(\mathsf{C})$ and thus polynomial in the size of C. A rule application always adds and never removes concepts from node labels and, after a rule has been applied to a node for a concept, it is not applicable anymore. A new node can only be created by the $\exists$-rule, and the concepts that are added to its label (by the $\forall$- and $\exists$-rule) are shorter than the concepts in the father node's label, which implies that the depth of the completion tree is linearly bounded by the size of the input. There is also a limitation on the width of the tree because, for every node, at most one child node can be created for every existential restriction in $sub(\mathsf{C})$. Thus, by König's lemma, the algorithm generates a finite tree whose nodes are labelled with finite sets. □

Regarding space complexity, observe that we only need to store one path in memory at a time if we traverse the tree depth-first, free the memory when returning to a predecessor node, and remember, for each node along the path, which successors have already been processed. Since the depth of the tree and each node label (including the required backtracking information) are linearly bounded by the size of the input, we obtain that $\mathcal{ALC}$ concept satisfiability is in PSPACE. PSPACE-hardness can be shown by a reduction from the satisfiability problem of Quantified Boolean Formulas (Schmidt-Schauß and Smolka, 1991), thus we obtain the following:

Corollary 3.3. Concept satisfiability for $\mathcal{ALC}$ is PSPACE-complete.

The algorithm described above shows mainly the positive properties of TAs mentioned in Chapter 1: firstly, since $\mathcal{ALC}$-satisfiability is PSPACE-hard and the TA is a PSPACE algorithm, it provides a tight upper bound. Secondly, it is goal-directed because every concept that is added by a rule application brings the algorithm closer to finding a model or to showing that there can be no model. For the "mother" example in Equation 1.1 on page 7, the TA terminates after three rule applications: two applications of the $\sqcap$-rule add the three conjuncts to the root node, then the $\exists$-rule creates a $\mathsf{has\text{-}child}$-successor labelled with Human, after which the completion tree is saturated. Since it is also clash-free, we have obtained a tableau. Thirdly, the algorithm is amenable to the known optimisations for tableau algorithms. In the following, we will sketch two of the most efficient such optimisations, which deal with branching and backtracking, i. e. with the handling of non-determinism.

Backjumping. After the detection of a clash, the default algorithm for backtracking goes back to the most recent non-deterministic decision for which there exists another alternative. *Backjumping* (Baker, 1995; Horrocks, 1997) goes back to the most recent decision *one of the clashing concepts depends on*, skipping over those ones that did not have any influence on the clash. For example, if a node label contains the concepts $(\mathsf{A} \sqcup \mathsf{B}), (\mathsf{D_1} \sqcup \mathsf{E_1}), \ldots, (\mathsf{D_n} \sqcup \mathsf{E_n}), (\neg\mathsf{A} \sqcap \neg\mathsf{C})$, the $\sqcup$-rule might first add A, then $\mathsf{D}_1, \ldots, \mathsf{D}_n$ and finally $\neg\mathsf{A}$ and $\neg\mathsf{C}$, causing a clash. The naive backtracking algorithm would first try to add E_n instead of D_n, then E_{n-1} instead of D_{n-1} and so on, thus testing 2^n different combinations of D_i and E_i concepts before changing the decision that can actually remedy the clash. With backjumping, the concept A is labelled with

"{1}" to indicate that it depends on the first non-deterministic decision, and $\neg \mathsf{A}$ is labelled with $\emptyset$ since it does not depend on any decision. When the clash between these concepts occurs, the algorithm immediately goes back to the first decision, jumping over the n intermediate ones.

Backjumping requires additional time and space in order to store the dependency information, but its positive effects significantly outweigh this overhead (Horrocks and Patel-Schneider, 1999; Hladik, 2002).

Semantic Branching. After backtracking due to a clash involving a concept C that was added by the $\sqcup$-rule for a disjunction $\mathsf{C} \sqcup \mathsf{D}$, the naive (*syntactic*) branching algorithm simply adds D. Thus the information that C is unsatisfiable in the current node is lost, and later on the $\sqcup$-rule might add C again for another disjunction, say $\mathsf{C} \sqcup \mathsf{E}$. In order to make the implicit information that C is unsatisfiable explicit, the *semantic* branching heuristics, which was adapted from the DPLL procedure for propositional logic (Davis et al., 1962), adds $\neg\mathsf{C} \sqcap \mathsf{D}$ after a clash caused by adding C. This technique aims at pruning the search space since a completion tree in which C can be added to the corresponding node is never tested again.

Like backjumping, semantic branching introduces an overhead, in this case by adding the concept $\neg\mathsf{C}$. If C is a concept name, this may be negligible, but if adding $\neg\mathsf{C}$ necessitates the creation of additional nodes or leads to further non-determinism, the time required for processing $\neg\mathsf{C}$ might be longer than the time saved by avoiding a possible later addition of C. Empirical results show, however, that although a slight performance decrease may occur for some concepts, semantic branching leads to a significant speedup in the vast majority of the test cases (Horrocks and Patel-Schneider, 1999; Hladik, 2002).

3.2.2 A Tableau Algorithm for $\mathcal{ALC}$ with General TBoxes

The tableau algorithm and its correctness proof become more difficult when testing the satisfiability of a concept C w. r. t. a TBox $\mathcal{T}$. Now we require an additional rule to deal with GCIs (see Figure 3.3), and the node labels can also contain concepts appearing in $\mathcal{T}$, which is the reason why we have to modify the definition of the completion tree.

Definition 3.4 (Completion tree for $\mathcal{ALC}$ with GCIs). For a concept C and a TBox $\mathcal{T}$, $\mathit{sub}(\mathsf{C}, \mathcal{T})$ is defined as follows:

$$\mathit{sub}(\mathsf{C}) \quad \cup \bigcup_{\mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}} \mathit{sub}(\doteq\neg\mathsf{D} \sqcup \mathsf{E})$$

Similarly, $\mathit{rol}(\mathsf{C}, \mathcal{T})$ is the set of all role names appearing in C or $\mathcal{T}$. A *completion tree for* C *w. r. t.* $\mathcal{T}$ is a tree whose nodes are labelled with subsets of $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and whose edges are labelled with edges from $\mathit{rol}(\mathsf{C}, \mathcal{T})$. $\diamond$

In this case, the size of each node label is still polynomially bounded by the size of the input $(\mathsf{C}, \mathcal{T})$, but the maximum size of the concepts contained in a node label does

$\mathcal{T}$**-rule** **If** there is a GCI $\mathsf{C} \sqsubseteq \mathsf{D} \in \mathcal{T}$
and there is a node v with $(\doteq\neg\mathsf{C} \sqcup \mathsf{D}) \notin n(v)$
then set $n(v) := n(v) \cup \{\doteq\neg\mathsf{C} \sqcup \mathsf{D}\}$.

$\exists$**-rule** **If** for a *non-blocked* node v, $n(v)$ contains a concept $\exists\mathsf{r}.\mathsf{C}$
and there is no r-successor w of v with $\mathsf{C} \in n(w)$,
then create a new r-successor w of v with $n(w) = \{\mathsf{C}\}$.

Figure 3.3: Modifications of the $\mathcal{ALC}$ tableau algorithm to handle GCIs

not decrease with the depth of the node, as in Section 3.2.1, since it never falls below the level of the concepts contained in the GCIs. For example, for $\mathsf{C} \sqsubseteq \exists\mathsf{r}.\mathsf{C}$, the rules from Figure 3.2 would create an infinite chain of nodes if C appears in the completion tree. However, since the nodes can only be labelled with sets of subconcepts of C and $\mathcal{T}$, we have a bound on the number of nodes with *different* labels. In order to ensure termination, we have to avoid the creation of different nodes with the same label (Buchheit, Donini, and Schaerf, 1993; Baader, Buchheit, and Hollunder, 1996).

Definition 3.5 (Blocked). Let $T = (V, E, n, \ell)$ be a completion tree. A node $v \in V$ is called *directly blocked* if there is a predecessor node u of v such that $n(v) = n(u)$ and u is not blocked; v is called *indirectly blocked* if one of v's predecessors is directly blocked; and v is *blocked* if it is directly or indirectly blocked.[2] ◇

We then modify the tableau algorithm by applying the $\exists$-rule only to non-blocked nodes, see Figure 3.3. This way, we immediately obtain a new termination result since there is only a finite number of different node labels possible. However, the soundness proof becomes more difficult: when we try to find a witness for the satisfiability of an existential restriction in a blocked node, we either have to introduce a relation between the predecessor of a blocked node and the corresponding blocking node (as in the proof for Theorem 3.6 below), or we have to create a copy of the blocking node and its successors in place of the blocked node. Arguing why this procedure does not lead to contradictions makes many of these proofs rather intricate, particularly in the presence of inverse roles and number restrictions (see e. g. Horrocks et al., 2000a).

From a model-theoretic point of view, the reason why blocking is required to ensure termination in one case and not in the other one is that $\mathcal{ALC}$ without general TBoxes has the *finite tree model property*, i. e. every satisfiable concept has a model which is a finite tree. $\mathcal{ALC}$ with general TBoxes has the *finite model property*, but a finite model might be cyclic, and the *tree model property*, but a tree model might be infinite, and using the rules from Figure 3.2, the TA attempts to build such a possibly infinite model.

[2] In the case of $\mathcal{ALC}$, it is possible to use *subset blocking*, i. e. "$n(v) \subseteq n(u)$" in the definition of the blocking relation. However, for the sake of simplicity and consistency with blocking relations for more expressive logics, e. g. in Chapter 7, we use the more general notion of *equality blocking* in Definition 3.5.

Theorem 3.6. The tableau algorithm for $\mathcal{ALC}$ with general TBoxes effectively decides satisfiability.

Proof. We again show soundness, completeness and termination.

Soundness. Let (V, E, n, ℓ) be a tableau for C w. r. t. $\mathcal{T}$. We generate a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ as follows:

- $\Delta^{\mathcal{I}} := \{v \in V \mid v \text{ is not blocked}\}$;
- For a concept name D, $\mathsf{D}^{\mathcal{I}} := \{v \in \Delta^{\mathcal{I}} \mid \mathsf{D} \in n(v)\}$;
- For a role name r, $\mathsf{r}^{\mathcal{I}} := \{(v, w) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid$ such that one of the two following conditions is satisfied:
  - $(v, w) \in E$, $\ell(v, w) = \mathsf{r}$, or
  - $(v, x) \in E$, $\ell(v, x) = \mathsf{r}$, for some node x such that x is blocked by w.$\}$

It follows by induction over the structure of concept terms that the individual v satisfies all concept in the label of the node v and all GCIs in $\mathcal{T}$:

- For concept names and Boolean operators, this follows as in the case without GCIs (Theorem 3.2).
- For an existential restriction $\exists \mathsf{r}.\mathsf{D} \in n(v)$ for a non-blocked node v, since the tableau is saturated, there exists an r-successor x of v labelled with D. If x is not blocked, then x belongs to $\Delta^{\mathcal{I}}$, and thus it also holds that $(v, x) \in \mathsf{r}^{\mathcal{I}}$ and $x \in \mathsf{D}^{\mathcal{I}}$. Otherwise, there is a node w that blocks x. By construction, $r^{\mathcal{I}}$ contains (v, w) and, since $n(x) = n(w)$ and w is not blocked, w belongs to $\mathsf{D}^{\mathcal{I}}$.
- For a value restriction $\forall \mathsf{r}.\mathsf{D}$, assume there is an individual w such that $(v, w) \in \mathsf{r}^{\mathcal{I}}$. Then w is either an r-successor of v in the tableau, in which case $n(w)$ contains D because the tableau is saturated, or there is an r-successor x of v such that w blocks x. Then $n(x)$ contains D and, since $n(w) = n(x)$ holds, w belongs to $\mathsf{D}^{\mathcal{I}}$ by induction hypothesis.
- For a GCI $\mathsf{D} \sqsubseteq \mathsf{E}$, assume $v \in \mathsf{D}^{\mathcal{I}}$ (otherwise there is nothing to show). As no rule is applicable, $n(v)$ contains $\dot{\neg}\mathsf{D} \sqcup \mathsf{E}$, and thus $\dot{\neg}\mathsf{D}$ or E. If v contains $\dot{\neg}\mathsf{D}$ then, by the induction above, $v \in (\dot{\neg}\mathsf{D})^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \mathsf{D}^{\mathcal{I}}$, which contradicts our assumption. Otherwise, $n(v)$ contains E, which implies $v \in \mathsf{E}^{\mathcal{I}}$, and thus v satisfies the GCI.

Completeness. We will again show how a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for C w. r. t. $\mathcal{T}$ can be used to find a sequence of rule applications that leads to a clash-free and saturated completion tree (V, E, n, ℓ) by defining a function φ as in the proof for Theorem 3.2. We only present the arguments for the rules in Figure 3.3 since the other ones do not require modifications.

The $\exists$-rule: Let v be a node with $\exists \mathsf{r}.\mathsf{D} \in n(v)$. If v is blocked, we have nothing to show. Otherwise, the existence of an appropriate r-successor follows as before.

The $\mathcal{T}$-rule: For a GCI $\mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}$, it holds that every individual in $\mathsf{D}^{\mathcal{I}}$ is also a member of $\mathsf{E}^{\mathcal{I}}$, i. e. it belongs to $\mathsf{E}^{\mathcal{I}}$ or to $(\neg\mathsf{D})^{\mathcal{I}}$, and thus also to $(\dot{\neg}\mathsf{D} \sqcup \mathsf{E})^{\mathcal{I}}$. We can therefore add $\dot{\neg}\mathsf{D} \sqcup \mathsf{E}$ to the label of every node $v \in V$ without violating the conditions.

Termination. Each node label is a subset of the set $\mathit{sub}(\mathsf{C}, \mathcal{T})$, whose size is polynomial in the size of C and $\mathcal{T}$. Since we never remove any concepts from node labels, the number of rule applications for a single node is bounded by the size of $\mathit{sub}(\mathsf{C}, \mathcal{T})$, i. e. it is polynomial. The width of the tree is linearly bounded by the size of C and $\mathcal{T}$, similarly to the case of concept satisfiability (Theorem 3.2). However, there can be an exponential number of different node labels on a path before a node on the path is blocked, which means that the completion tree we construct can have a depth that is exponential in the size of the input. The number of nodes in a tree can be exponential in its depth, which leads to a double-exponential number of possible nodes. Since the algorithm is non-deterministic, we obtain that the algorithm is in 2-NExpTime. □

For this algorithm, the negative features of tableau algorithms also begin to show. Most importantly, the 2-NExpTime result is far from optimal considering that concept satisfiability w. r. t. general TBoxes is ExpTime-complete (see Section 2.3.6). Here, the estimate of an exponential number of unblocked nodes on a path may seem overly pessimistic, but in fact it is easy to construct an example in which this behaviour can actually occur: when testing the satisfiability of a concept name A w. r. t. the GCI $\top \sqsubseteq \exists \mathsf{r}.\top \sqcap (\mathsf{A}_1 \sqcup \mathsf{B}_1) \sqcap (\mathsf{A}_2 \sqcup \mathsf{B}_2) \sqcap \ldots \sqcap (\mathsf{A}_n \sqcup \mathsf{B}_n)$, there can be 2^n unblocked nodes with different labels on a path.

However, it is possible without major modifications to obtain a NExpTime result by changing the blocking condition in such a way that a node can not only be blocked by a predecessor within the tree, but also by other nodes. In this case, it must be ensured that two nodes cannot mutually block each other, thus preventing successors for both nodes from being created. This can be achieved by defining a total order on the nodes and permitting nodes to be blocked by predecessors with respect to that order (Baader et al., 1996). This way, the number of nodes in the entire completion tree is exponentially bounded by the size of the input.

Obtaining an ExpTime result is much more involved. The worst-case optimal tableau algorithm developed by Donini, De Giacomo, and Massacci (1996) requires remembering satisfiability information gathered for different nodes in the current completion tree (via blocking), and additionally remembering constellations that will inevitably lead to a clash, i. e. *unsatisfiability information*, which was gathered previously from completion trees resulting from different non-deterministic decisions. In the vocabulary of Section 3.2, this means that satisfiability information is propagated between nodes of the inner tree, whereas unsatisfiability information is propagated between different nodes of the outer tree. Thus, at most an exponential number of types for nodes is ever considered during the run of the TA. The algorithm becomes much more involved than the one sketched in this section, and the same holds for the proofs. From a practical point of view, it is questionable whether the improvement on

the worst-case complexity will also lead to a better performance in an implementation since the algorithm relies on introducing a significant amount of non-determinism. Although the methods by Donini et al. (1996) have recently been reconsidered for reasoning in expressive DLs (Goré and Nguyen, 2007; Ding and Haarslev, 2007), their influence on practical efficiency is still an open question.

The proof of Theorem 3.6 also shows how the blocking condition complicates the soundness proof since the definition of the model $\mathcal{I}$ treats blocked and unblocked notes differently, and so does the argument for the satisfaction of existential restrictions. For more expressive logics, further problems arise: in the presence of qualifying number restrictions and inverse roles, e. g. for the DL $\mathcal{SHIQ}$, it is no longer possible to "bend" the edges leading to blocked nodes back to the blocking nodes (thus transforming a tableau with blocked nodes into a finite model that is not tree-shaped anymore) because these additional edges might violate number restrictions in the blocking node. Instead, it is necessary to *unravel* the completion tree, i. e. to create a copy of the blocking node and all its successors, and thus transforming the finite tableau with blocked nodes into an infinite tree-shaped model. Moreover, since in the presence of inverse roles also the father node has an influence on the satisfaction or violation of number restrictions, it is necessary to define the blocking relation not between nodes, but between *pairs* of nodes (Horrocks et al., 2000a).[3]

With the blocking conditions becoming more complex, a tradeoff arises between a blocking condition that is easy to handle in the proofs and one that will lead to the best performance in practice. Horrocks and Sattler (2002) develop a blocking condition that is as general as possible, i. e. it aims at detecting blocked nodes at a very early stage, which promises to lead to a more efficient algorithm in practice, but requires very involved arguments in the proofs.

[3]The issue of the size of the structures used in the blocking condition is addressed in detail in Section 7.1 under the name of *pattern depth*.

Chapter 4

Automata Algorithms

Finite automata, which were introduced by Kleene (1956), are among the most basic machine models: they provide a finite memory to remember the current state, a way of switching from one state to another depending on the input, and only two possible outputs, namely *accepting* or *rejecting*, which is determined by the last state that the automaton reaches after reading the entire input. If the state belongs to a set of *final states*, the input is accepted, otherwise it is rejected.

In comparison to other machine models (e. g. Turing machines, *while*-programs, RAM machines), automata are restricted in the sense that they only allow a finite memory (state space), independent of the size of the input. Yet they are powerful enough to perform a variety of different tasks in computer science, e. g. lexical analysis of formal languages for compilers, natural language processing, or modelling of parallel processes (Perrin, 1990). In contrast to the more powerful machine models mentioned above, important decision problems, like the emptiness problem, are decidable and even tractable, i. e. they require only polynomial time in the size of the input.

In this chapter, we first give a brief introduction to automata in general, then we introduce two kinds of automata that are frequently used in the area of description logics, namely *non-deterministic* and *alternating tree automata*, and we give examples for automata algorithms deciding satisfiability for $\mathcal{ALC}$ concepts w. r. t. acyclic TBoxes.

4.1 Finite Automata

Formally, for a given *alphabet* Σ (representing the elements of the input), an automaton $\mathcal{A}$ consists of a finite set of *states* Q (the internal memory), a *transition relation* $\Delta \subseteq Q \times \Sigma \times Q$, a subset $I \subseteq Q$ of *initial* states and another subset $F \subseteq Q$ of *final* states. The intuition behind a transition (q_1, σ, q_2) is that $\mathcal{A}$, if it is in state q_1 and reads the letter σ, can switch to state q_2 (and continue the computation with the next letter of the input). A sequence of such transitions is called a *run*. A word over the alphabet is *accepted* if, through reading the entire word from left to right, the automaton can go from an initial to a final state according to the transition relation.

The automata introduced by Kleene (1956) operate on words, i. e. finite sequences of symbols, which is not well-suited for structures in mathematical or computational logic since these are often tree-shaped and also can be infinite. Büchi (1960) generalised Kleene's automata to infinite words by redefining the acceptance condition, and Rabin (1969) further generalised Büchi automata from words (i. e. unary trees) to k-ary trees. These tree automata lend themselves to the use in satisfiability checking for logics with the tree model property. The transition relation of a tree automaton is a subset of $Q \times \Sigma \times Q^k$, where k is the tree arity. The intuition is that the automaton continues by operating on every child of the current node in the corresponding state. Since this means that there are effectively k automata operating on the tree after the transition, the notion of *switching* to a state does not make sense anymore, and thus we will say that the automaton *sends copies* of itself to the children nodes.

Automata on infinite words and trees share many properties with automata on finite trees: a finite set of states, a subset of initial states, and a transition relation. However, since there is no last state after reading the entire input, the acceptance condition sketched above for finite objects is not applicable, and several new conditions were suggested. In the most basic version, called *looping automata*, an input is accepted if the automaton does not reach a "dead end", i. e. a state for which there is no successor state. *Büchi* automata again introduce a set of final states and accept an input if a final state is reached infinitely often while reading the input. This allows for the modelling e. g. of "dangerous" and "safe" states, where it is required that after each dangerous state, a safe state is reached eventually. *Rabin* automata additionally allow for the restriction that certain states must *not* appear infinitely often.

It is easy to see that looping automata can be regarded as special Büchi automata (where every state is final), and also that Büchi automata are a special case of Rabin automata (without forbidden states). It turns out that these inclusions are strict: Rabin automata are more powerful than Büchi ones (Rabin, 1970), which in turn are more powerful than looping automata (since looping automata are not closed under complement, i. e. the complement of a recognisable language is not necessarily recognisable). For the logics considered in this thesis, looping automata are sufficient, thus we will focus on this variant in the following. However, there are description logics for which automata with more sophisticated acceptance conditions are useful, e. g. DLs with fixpoint operators (Calvanese, De Giacomo, and Lenzerini, 1999; Sattler and Vardi, 2001) or with a transitive closure operator on roles (see e. g. Baader, 1991), which can be handled similarly to the star operator in **PDL** (Vardi and Wolper, 1986).

In order to decide the emptiness problem of looping automata on infinite structures, one computes the set $Q_\perp$ of states that cannot appear in an infinite run: $Q_\perp$ is initialised with those states from which there is no transition, and it is then extended with those states from which every transition involves a state $q \in Q_\perp$, and this procedure is iterated until no further states can be added. If all initial states are contained in $Q_\perp$ at the end of the iteration, the language accepted by the automaton is empty (because every sequence of transitions from an initial state leads to a dead end). Clearly, this test is polynomial in the size of Q, as there are at most $\#Q$ iterations during which at most $\#Q$ states are tested. For Büchi automata, the emptiness test

is also polynomial (Vardi and Wolper, 1986); it is NP-complete for Rabin automata (Emerson and Jutla, 1988).

If the transition relation is functional (i. e. there is only one transition from a state and letter), the automaton is called *deterministic* because there is at most one possible run for a specific input. *Non-deterministic* automata with several possible transitions allow for different runs on one input and, by definition, the automaton accepts an input if there exists an accepting run. Thus, the different transitions from a specific state reading a specific letter can be regarded as different alternatives, i. e. a disjunction of possibilities. *Alternating* automata are a further generalisation additionally allowing conjunction, i. e. sending several copies operating in different states to the same node.

In our notation of automata, we will use Δ for a transition relation and δ for a function. Moreover, we will omit the set of final states F in the case of looping automata and, if there is only one initial state q_0, we will write (Q, Σ, q_0, Δ) instead of $(Q, \Sigma, \{q_0\}, \Delta)$.

4.2 Automata for Description Logics

Testing satisfiability of a DL expression E (e. g. a single concept or a concept and a TBox) using tree automata is a two-step process: firstly, translating E into an automaton $\mathcal{A}_\mathsf{E}$ that accepts all models for E as input, and secondly, testing the emptiness of $\mathcal{L}(\mathcal{A}_\mathsf{E})$, the language accepted by $\mathcal{A}_\mathsf{E}$.[1] The idea is that the automaton recognises tree models for the input, where each node represents an individual: the root node stands for an individual satisfying the input, and the successors of a node v stand for individuals related with v via a role. Thus, the input trees look similar to completion trees generated by tableau algorithms, with the difference that the input trees have a fixed arity, which is why input trees can contain "dummy" nodes, i. e. nodes that do not correspond to an individual in a model. Moreover, the edges in an automata input tree are not labelled (the information about the role connecting two nodes is not contained in an edge label, but in the transition relation, see Section 4.3 below). Also, since many DLs have the tree model property, but not the finite tree model property, the input trees have an infinite depth, unlike completion trees.

In the case of non-deterministic automata, the first step, i. e. the translation of a DL input into an automaton, usually requires time which is exponential in the size of E, whereas the second step, i. e. the emptiness test described above, only requires polynomial time. For alternating automata, the translation usually requires only polynomial time (see e. g. Vardi, 1998; Calvanese et al., 1999), but the emptiness test involves a translation of the alternating automaton into a non-deterministic automaton of exponential size (Kupferman and Vardi, 1998; Vardi, 1998), thus the emptiness test again requires time exponential in the size of E. For ExpTime logics, this does not seem too bad considering that the worst-case complexity of the intuitive tableau algorithm usually is NExpTime (or 2-NExpTime, see Section 3.2.2) but with automata algorithms the exponential step has to be performed in *every* case, i. e. the

[1] In the following, we will sometimes use the expression "emptiness of $\mathcal{A}$" as an abbreviation for "emptiness of the language accepted by $\mathcal{A}$".

worst-case complexity really is the any-case complexity. Therefore, one cannot expect acceptable performance from a naive implementation that works by first creating the entire automaton and then testing its emptiness.

On the other hand, termination of the satisfiability test is not an issue, even if the underlying DL does not have the finite model property, because the emptiness test is performed on the finite automaton rather than the infinite inputs of the automaton. This makes automata attractive from the theoretical point of view, since the absence of a blocking condition simplifies the necessary proofs significantly.

4.3 Non-Deterministic Tree Automata

We will now formally introduce the notation of non-deterministic tree automata and the data structures they operate on, namely infinite k-ary trees, whose nodes are labelled with Hintikka sets (see Section 3.1), i. e. contradiction-free and propositionally expanded sets of concepts. We identify the nodes in a k-ary tree by words over the alphabet $\{1, \ldots, k\}$ in the following way: the empty word ε denotes the root node, and the i-th successor of a node v is identified by $v \cdot i$ for $1 \leq i \leq k$.

Definition 4.1 (Set K). For a natural number k, let K be the set $\{1, \ldots, k\}$. For a set K, we denote by K^* the set of all words over the alphabet K, and by K^+ we denote K^* without the empty word ε. ◇

Thus, the set of all nodes in a k-ary tree is K^* and, in the case of labelled trees, we will refer to the labelling of the node v in the tree t by $t(v)$.

Definition 4.2 (Non-deterministic automaton, run, accepted language). A *non-deterministic tree automaton (NTA)* over k-ary trees is a tuple (Q, Σ, I, Δ), where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q^k$ is a transition relation, and $I \subseteq Q$ is a set of initial states.

A *run* of an automaton $\mathcal{A} = (Q, \Sigma, I, \Delta)$ on a k-ary tree t is a labelled k-ary tree r such that, for all $v \in K^*$, it holds that:

$$(r(v), t(v), r(v \cdot 1), \ldots, r(v \cdot k)) \in \Delta$$

A run is called *successful* if $r(\varepsilon) \in I$. The *language accepted by* $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of all trees t such that there exists a successful run of $\mathcal{A}$ on t. ◇

The idea behind this definition is that the automaton starts in an initial state on the root node, chooses from the transition relation a transition compatible with the current state and the tree label, and then sends copies of itself to the successor nodes, where the state of each copy is determined by the transition. If there is a transition for every node of t, then t is accepted. The non-determinism results from the fact that Δ may allow several transitions for a pair $(r(v), t(v))$.

It is obvious from this definition that the successful run r is isomorphic with the tree t in the sense that both are k-ary and that the node v of the run refers to the node v in the tree (i. e. if $r(v) = q$, then the automaton is in state q and reads $t(v)$), thus the run is essentially a relabelling of the input tree.

In the following section, we introduce an automata algorithm for $\mathcal{ALC}$. Algorithms for other logics were developed e. g. by Lutz and Sattler (2000) and Lutz (2002).

4.3.1 An NTA Algorithm for $\mathcal{ALC}$ with General TBoxes

When using NTAs for the satisfiability test of a DL expression E, the input trees are candidates for models for E, with every node representing an individual of the domain. Due to the definition of the transition relation, the automaton has to descend within the tree with every transition, it cannot stay in the same node or go back to a lower depth. Consequently, every node is processed by only one state, which means that we have to check all concepts that the corresponding individual has to satisfy using this state—unlike in tableaus, where we can use several steps to expand the node label propositionally. Moreover, we do not need to consider nodes whose labels contain obvious contradictions (clashes) because in a model every individual either belongs to $\mathsf{C}^{\mathcal{I}}$ or to $(\neg\mathsf{C})^{\mathcal{I}}$ for a concept C. For this reason, it is useful not to allow random sets of concepts in the node labels, but rather only propositionally expanded and clash-free sets, i. e. Hintikka sets. Like in the previous chapter, we will assume for the sake of simplicity that all concepts are in NNF.

Definition 4.3 (Hintikka set). Let C be an $\mathcal{ALC}$ concept and $\mathcal{T}$ be a TBox. A set $H \subseteq \mathit{sub}(\mathsf{C}, \mathcal{T})$ is called a *Hintikka set* if the following three conditions are satisfied:

- if $\mathsf{D} \sqcap \mathsf{E} \in H$, then $\{\mathsf{D}, \mathsf{E}\} \subseteq H$;
- if $\mathsf{D} \sqcup \mathsf{E} \in H$, then $\{\mathsf{D}, \mathsf{E}\} \cap H \neq \emptyset$;
- there is no concept name A with $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq H$.

A Hintikka set H is called $\mathcal{T}$*-expanded* if, for every GCI $\mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}$, it holds that $\doteq\neg\mathsf{D} \sqcup \mathsf{E} \in H$. ◇

Note that $\#\mathit{sub}(\mathsf{C}, \mathcal{T})$ is polynomial in the length of C and $\mathcal{T}$. The set $\mathit{sub}(\mathsf{C}, \mathcal{T})$ contains the concept C and one concept for each GCI; and the number of subconcepts for each of those concepts is linear in their length.

We can now represent a tree model for an $\mathcal{ALC}$ concept C and TBox $\mathcal{T}$ as a tree whose nodes are labelled with Hintikka sets: a node represents an individual, the node label contains the concepts to which the corresponding individual belongs, and the edges between a node and its children stand for the roles by which they are related. Since our trees do not contain edge labels, we handle the edge labels implicitly by enumerating all existential concepts in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and defining that the leftmost child stands for the first existential restriction, the second child stands for the second existential restriction, etc.

Definition 4.4 (Hintikka tree). For a concept C and TBox $\mathcal{T}$, fix an ordering of the existential concepts in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and let $\varphi : \{\exists\mathsf{r}.\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T})\} \to K$ be the corresponding ordering function. Then, the tuple $(\Omega_0, \Omega_1, \ldots, \Omega_k)$ is called $\mathsf{C}, \mathcal{T}$*-compatible* if, for all $0 \leq i \leq k$, Ω_i is a $\mathcal{T}$-expanded Hintikka set and, for every existential restriction $\exists\mathsf{r}.\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T})$ with $\varphi(\exists\mathsf{r}.\mathsf{D}) = i$ it holds that

- if $\exists \mathsf{r}.\mathsf{D} \in \Omega_0$, then
  1. Ω_i contains D;
  2. Ω_i contains all concepts E_j for which there is a value restriction $\forall \mathsf{r}.\mathsf{E}_j \in \Omega_0$;
- if $\exists \mathsf{r}.\mathsf{D} \notin \Omega_0$, then $\Omega_i = \#$.

Moreover, the tuple where $\Omega_0 = \Omega_1 = \ldots = \Omega_k = \#$ holds is also $\mathsf{C}, \mathcal{T}$-compatible. Nodes labelled with $\#$ are called *dummy* nodes because they are not required in a model. A k-ary tree t is called a *Hintikka tree for* C *and* $\mathcal{T}$ if, for every node $v \in K^*$, the tuple $(t(v), t(v \cdot 1), \ldots, t(v \cdot k))$ is $\mathsf{C}, \mathcal{T}$-compatible. For a role r, we say that a node w is an r-*successor* of a node v if $w = v \cdot \varphi(\exists \mathsf{r}.\mathsf{D})$ for some concept D and $\exists \mathsf{r}.\mathsf{D} \in \Omega(v)$.$\diamond$

Note that in a Hintikka tree t, the i-th successor of a node v stands for the individual satisfying the i-th existential restriction D if $\mathsf{D} \in t(v)$. The definition of $\mathsf{C}, \mathcal{T}$-compatibility ensures that value restrictions are handled correctly. Using this property, we can show that the existence of Hintikka trees characterises satisfiability in $\mathcal{ALC}$:

Theorem 4.5. An $\mathcal{ALC}$ concept C is satisfiable w. r. t. a TBox $\mathcal{T}$ iff there is a $\mathsf{C}, \mathcal{T}$-compatible Hintikka tree t with $\mathsf{C} \in t(\varepsilon)$.

Proof. For the "if" direction, we will show how to construct a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ from t. Let $\Delta^{\mathcal{I}} = \{v \in K^* \mid t(v) \neq \#\}$. For a role name $\mathsf{r} \in \mathsf{N_R}$, we define $\mathsf{r}^{\mathcal{I}} = \{(v, w) \mid w \text{ is an } \mathsf{r}\text{-successor of } v\}$. For a concept name A, we define $\mathsf{A}^{\mathcal{I}} = \{v \in \Delta^{\mathcal{I}} \mid \mathsf{A} \in \Omega(v)\}$. Then it follows by structural induction that, for every concept D with $\mathsf{D} \in \Omega(v)$, it holds that $v \in \mathsf{D}^{\mathcal{I}}$:

- if $\mathsf{E} \sqcap \mathsf{F} \in \Omega(v)$ then, since $\Omega(v)$ is a $\mathsf{C}, \mathcal{T}$-expanded Hintikka set, it contains E and F, and by induction $v \in \mathsf{E}^{\mathcal{I}} \cap \mathsf{F}^{\mathcal{I}}$ holds,
- if $\mathsf{E} \sqcup \mathsf{F} \in \Omega(v)$ then $v \in \mathsf{E}^{\mathcal{I}} \cup \mathsf{F}^{\mathcal{I}}$ follows from an analogous argument,
- if $\exists \mathsf{r}.\mathsf{E} \in \Omega(v)$ for a role name r then, since t is a Hintikka tree, $(v, v \cdot \varphi(\exists \mathsf{r}.\mathsf{E})) \in \mathsf{r}^{\mathcal{I}}$ and $\mathsf{E} \in \Omega(v \cdot \varphi(\exists \mathsf{r}.\mathsf{E}))$, thus $v \in (\exists \mathsf{r}.\mathsf{E})^{\mathcal{I}}$,
- if $\forall \mathsf{r}.\mathsf{E} \in \Omega(v)$ for a role r and $(v, w) \in \mathsf{r}^{\mathcal{I}}$ then, since $t(w) \neq \#$, there is a concept $\exists \mathsf{r}.\mathsf{F} \in t(v)$ such that $w = v \cdot \varphi(\exists \mathsf{r}.\mathsf{F})$. Then $\mathsf{E} \in \Omega(w)$ holds by definition of $\mathsf{C}, \mathcal{T}$-compatible.

For a GCI $\mathsf{E} \sqsubseteq \mathsf{F}$ from $\mathcal{T}$, $\Omega(v)$ contains $\doteq\neg\mathsf{E} \sqcup \mathsf{F}$ for every node v. As $\Omega(v)$ is a Hintikka set, it contains F or $\doteq\neg\mathsf{E}$. If it contains F then, as we have just shown, v belongs to $\mathsf{F}^{\mathcal{I}}$. Otherwise $\Omega(v)$ contains $\doteq\neg\mathsf{E}$, and $v \in (\doteq\neg\mathsf{E})^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \mathsf{E}^{\mathcal{I}}$ holds, which implies $v \notin \mathsf{E}^{\mathcal{I}}$. Therefore every node $v \in \mathsf{E}^{\mathcal{I}}$ is also contained in $\mathsf{F}^{\mathcal{I}}$.

For the "only-if" direction, we show how a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for C w. r. t. $\mathcal{T}$ can be used to define a $\mathsf{C}, \mathcal{T}$-compatible Hintikka tree t with $\mathsf{C} \in t(\varepsilon)$. Let k be the number of existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and φ be a function as in Definition 4.4. We

inductively define a function $\vartheta : K^* \to \Delta^{\mathcal{I}} \cup \{\mathsf{dummy}\}$ for a new individual dummy such that $t(v)$ contains all concepts that $\vartheta(v)$ satisfies.

Since $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model for C, there exists an element $d_0 \in \Delta^{\mathcal{I}}$ with $d_0 \in \mathsf{C}^{\mathcal{I}}$. Thus we set $\vartheta(\varepsilon) = d_0$ and $t(\varepsilon) = \{\mathsf{E} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid d_0 \in \mathsf{E}^{\mathcal{I}}\}$. Then we inductively define, for every node v for which ϑ is already defined, the labels of $v \cdot i$ with $i \in K$ as follows: if $t(v)$ contains the existential restriction $\exists \mathsf{r}.\mathsf{E}$ with $i = \varphi(\exists \mathsf{r}.\mathsf{E})$ then, since $\vartheta(v)$ satisfies $\exists \mathsf{r}.\mathsf{E}$, there exists an individual $d \in \Delta^{\mathcal{I}}$ with $(\vartheta(v), d) \in \mathsf{r}^{\mathcal{I}}$ and $d \in \mathsf{E}^{\mathcal{I}}$, and thus we set $\vartheta(v \cdot i) = d$, $t(v \cdot i) = \{\mathsf{F} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid d \in \mathsf{F}^{\mathcal{I}}\}$. If $\vartheta(v)$ does not belong to $(\exists \mathsf{r}.\mathsf{E})^{\mathcal{I}}$, we define $\vartheta(v \cdot i) = \mathsf{dummy}$ and $t(v \cdot i) = \#$.

It follows by construction that the tuple $(t(v), t(v{\cdot}1), \ldots, t(v{\cdot}k))$ is $\mathsf{C}, \mathcal{T}$-compatible. Note that for every $v \in K^*$, $t(v)$ is either $\#$ or a Hintikka set: since $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model, $d \in (\mathsf{E} \sqcup [\sqcap]\mathsf{F})^{\mathcal{I}}$ implies $d \in \mathsf{E}^{\mathcal{I}} \cup [\cap]\mathsf{F}^{\mathcal{I}}$, and $d \in \mathsf{E}^{\mathcal{I}}$ holds iff $d \notin (\neg \mathsf{E})^{\mathcal{I}}$ holds. □

With this result, we can use automata operating on Hintikka trees to test for the existence of models, i. e. to perform the satisfiability test for $\mathcal{ALC}$ concepts w. r. t. general TBoxes.

Definition 4.6 (Automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$). For an $\mathcal{ALC}$ concept C and a TBox $\mathcal{T}$ with k existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$, fix an ordering of these existential restrictions and let $\varphi : \{\exists \mathsf{r}.\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T})\} \to K$ be the corresponding ordering function. Then the looping automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, \Sigma, I, \Delta)$ is defined as follows:

- $Q = \Sigma = \{\Omega \in 2^{\mathit{sub}(\mathsf{C},\mathcal{T})} \mid \Omega \text{ is a } \mathcal{T}\text{-expanded Hintikka set}\} \cup \{\#\}$;
- Δ consists of all tuples $(\Omega_0, \Omega_0, \Omega_1, \ldots, \Omega_k)$ such that $(\Omega_0, \Omega_1, \ldots, \Omega_k)$ is $\mathsf{C}, \mathcal{T}$-compatible;
- $I = \{\Omega \in Q \mid \mathsf{C} \in \Omega\}$. ◇

Example 4.7. Figure 4.1 shows the automaton $\mathcal{A}_{\mathsf{Mother}}$ generated according to Definition 4.6 for the concept $\mathsf{Human} \sqcap \neg\mathsf{Male} \sqcap \exists \mathsf{has\text{-}child}.\mathsf{Human}$ from page 7 (concept and role names have been abbreviated). Transitions from one state to another are indicated by arrows. In order to avoid drawing 49 arrows in the upper half of the figure, we use the circle to indicate that there is a transition from every state with an arrow *toward* the circle (these are the states containing the concept $\exists \mathsf{has\text{-}child}.\mathsf{Human}$) to every state with an arrow *from* the circle (the states containing Human). Since there is only one existential subconcept, $\mathcal{A}_{\mathsf{Mother}}$ takes unary trees as input, i. e. it is in fact a word automaton.

Even in this very simple example, the overhead introduced by the exponential state space clearly shows:

- Out of the 14 states of the automaton, only one is required to verify the existence of a successful run, namely the initial state at the top, which has a transition to itself.
- Since there is a transition from every state, the emptiness test terminates after the first iteration ($Q_\perp$ is empty).

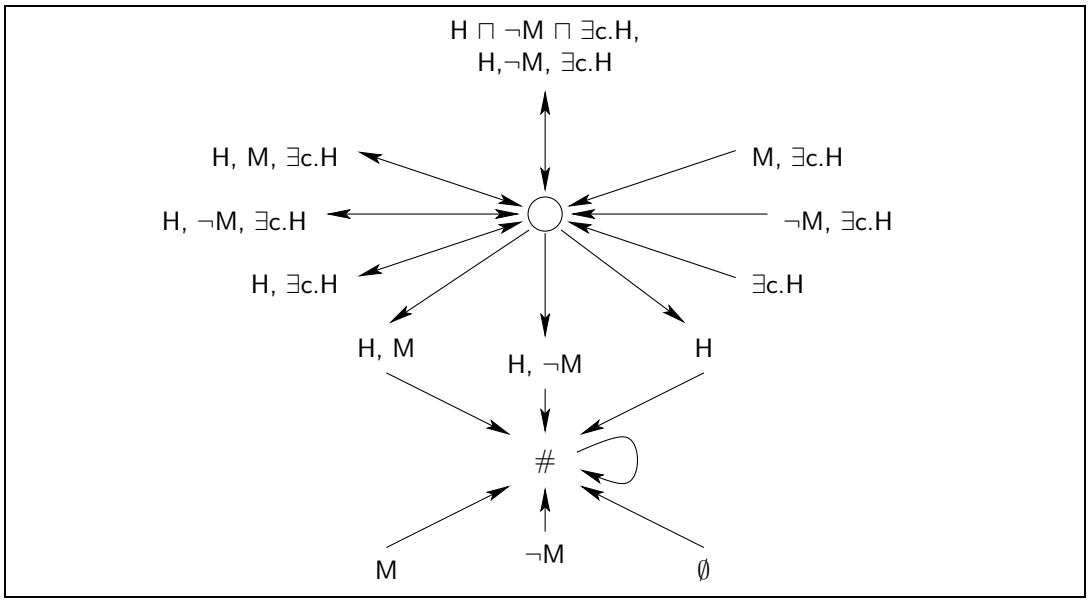

Figure 4.1: The NTA $\mathcal{A}_{\mathsf{Mother}}$ for the concept $\mathsf{H} \sqcap \neg\mathsf{M} \sqcap \exists\mathsf{c}.\mathsf{H}$

- Six states (those that do not contain Human) are not reachable from any other state and therefore irrelevant.

These properties are only detected after the creation of the whole automaton because emptiness is tested bottom-up. If one performed the emptiness test top-down and generated only those states that are required, one could in fact terminate the test after the first step, i. e. after detecting that the initial state has a transition to itself, since this gives rise to a successful run where every node is labelled with the initial state. However, this emptiness test would necessitate the introduction of a termination condition in order to avoid trying to actually generate an infinite run. Moreover, even after ensuring termination, the emptiness test would be non-deterministic (because the automaton is non-deterministic) and therefore only yield a non-deterministic complexity class.

Using $\mathcal{A}_{\mathsf{C},\mathcal{T}}$, we can reduce the satisfiability problem for $\mathcal{ALC}$ to the (non-) emptiness problem of $\mathcal{L}(\mathcal{A}_{\mathsf{C},\mathcal{T}})$. Since the size of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is exponential in the length of C and $\mathcal{T}$ (because there is one state for every possible Hintikka set of concepts) and the emptiness test of a looping automaton is polynomial in its size, we obtain an ExpTime algorithm for the satisfiability test of $\mathcal{ALC}$ concepts w. r. t. general TBoxes.

It is interesting to note that the alphabet symbol in each transition is practically redundant because it is always identical to the current state and also the sets Q and Σ are equal. In Chapter 6, we make use of this property by omitting Σ entirely and using automata on unlabelled trees.

Theorem 4.8. The language accepted by the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is empty iff C is unsatisfiable w. r. t. $\mathcal{T}$.

Proof. Since the transition relation Δ is defined exactly as the relation $\mathsf{C}, \mathcal{T}$-compatible, this follows by a simple induction. □

This theorem immediately yields a worst-case optimal upper bound for the corresponding decision problem.

Corollary 4.9. The satisfiability problem for $\mathcal{ALC}$ concepts w. r. t. general TBoxes is in ExpTime.

Proof. The automaton we constructed is exponential in the size of the input $\mathsf{C}, \mathcal{T}$. Since the emptiness of a looping automaton can be tested in time polynomial in the size of the automaton's state space (see the beginning of this section), we require time exponential in the size of the input. □

Like the automata algorithm itself, which first translates an input C into an automaton $\mathcal{A}_\mathsf{C}$ and then tests $\mathcal{A}_\mathsf{C}$ for emptiness, the proof of soundness and completeness of the automata algorithm is a two-step process: we first have to establish that the existence of Hintikka trees is a criterion for satisfiability, then we show how automata can be used to test for their existence.

This example for an automata-based decision procedure shows the positive and negative properties of AAs mentioned in Chapter 1. The algorithm from Definition 4.6 provides a tight upper complexity bound, and it handles termination and non-determinism implicitly: both the translation of the input $\mathsf{C}, \mathcal{T}$ into $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ and the emptiness test of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ are terminating and deterministic. The non-determinism related to constructors like $\sqcup$ is handled by the construction of the automaton, and termination results from the fact that the emptiness test is performed on the finite automaton rather than a possibly infinite model. This means that we do not have to define a blocking condition and deal with blocked nodes in the soundness proof. Thus from a theoretical point of view, the NTA algorithm is more elegant than the tableau one.

The disadvantages regarding the possibility for an implementation that displays acceptable performance in practice have already been pointed out in Example 4.7: since the search for a run proceeds in bottom-up direction and therefore is not goal-directed, it involves generating a significant amount of redundant information. Another drawback lies in the fact that the different constructors are handled with different mechanisms: conjunctions and disjunctions are dealt with by the definition of Hintikka sets, whereas existential and value restrictions are handled by the transition relation. Consequently, the connection between the constructors and the mechanisms dealing with them is not as direct as in the case of tableaus where, at least in the case of $\mathcal{ALC}$, there is a one-to-one correspondence between rules and constructors.

4.4 Alternating Tree Automata

A translation that is more similar to tableau rules and more intuitive can be achieved using alternating automata (Muller and Schupp, 1987). As hinted in Section 4.1, alternating automata do not only allow for a disjunction of alternatives, but also

for a conjunction or a combination of both. For example, the transition $\delta(\sigma, q_1) = (1, q_3) \wedge ((1, q_2) \vee (3, q_1))$ is to be read as follows: if the automaton is in state q_1 and operates on a node v that is labelled with the letter σ, then it sends one copy of itself in state q_3 to the first child of v and either another copy in state q_2 to the first child or a copy in state q_1 to the third one. This example illustrates that it is possible to send several copies to the same node and not to send any copies to others. This is reflected in the transition function by allowing for *positive Boolean formulas*, i. e. propositional formulas that do not contain negation.

As a further extension, alternating *two-way* automata (Vardi, 1989) additionally allow for the automaton to send copies to the current node or its predecessor. Thus, where the copies of an NTA as in Definition 4.2 march over the input tree in-line, with every copy operating on a node of depth n after n transitions, two-way automata can arbitrarily walk up and down on the input tree, and thus they can operate at any depth up to n. In order to describe such transitions, two-way automata do not only allow for the numbers in K in a transition, but additionally for 0 (representing the current node) and -1 (for the father node).

Definition 4.10 (Positive Boolean formula). Let k and K be as in Definition 4.1. We define $K_- := K \cup \{0, -1\}$. For a word $w = v \cdot c$ with $v \in K^*$ and $c \in K$, we define $w \cdot 0 := w$ and $w \cdot (-1) := v$; $\varepsilon \cdot (-1)$ is undefined.

The set of *positive Boolean formulas* over a set V, $\mathcal{B}^+(V)$, consists of formulas built from $V \cup \{\mathsf{true}, \mathsf{false}\}$ using the binary operators $\wedge$ and $\vee$. A set $R \subseteq V$ *satisfies* a formula $\varphi \in \mathcal{B}^+(V)$ if assigning true to all elements of R and false to all elements of $V \setminus R$ yields a formula that evaluates to true. ◇

Since non-determinism can be defined simply by using disjunctions, alternating automata have a transition *function* δ instead of a relation Δ. A non-deterministic automaton can be regarded as a special case of alternating automata in which every transition is in disjunctive normal form, with every conjunction having width k and every $c \in K$ appearing exactly once in every conjunction. For example, an NTA with the transitions $\{(q_0, a, q_1, q_2), (q_0, a, q_3, q_4)\} \subseteq \Delta$ can be translated into an ATA with $\delta(q_0, a) = ((1, q_1) \wedge (2, q_2)) \vee ((1, q_3) \wedge (2, q_4))$.

Definition 4.11 (Alternating automaton, run). An *alternating tree automaton (ATA)* $\mathcal{A}$ is a tuple (Q, Σ, q_0, δ), where Q is a set of states, Σ is the input alphabet, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \to \mathcal{B}^+(K_- \times Q)$ is the transition relation. The *width* of an automaton $w(\mathcal{A})$ is the number of literals that can appear on the right-hand side of a transition, i. e. $w(\mathcal{A}) := (\#Q + 1) \cdot (k + 2)$.

A *run* r of $\mathcal{A}$ on a tree t is a $w(\mathcal{A})$-ary infinite tree over $(K^* \times Q) \cup \{\#\}$ such that, for each node x with $r(x) = (v, q) \neq \#$ and $\delta(q, t(v)) = \varphi$, there is a set $S = \{(v_1, q_1), \ldots, (v_n, q_n)\} \subseteq K_- \times Q$ that satisfies the following conditions:

1. S satisfies φ and,

2. for all $1 \leq i \leq n$, $r(x \cdot i) = (v \cdot v_i, q_i)$.

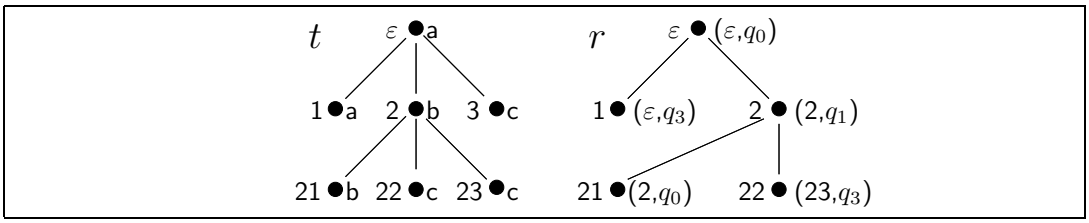

Figure 4.2: An input tree t and a successful run r of an alternating automaton

A run r is *successful* if $r(\varepsilon) = (\varepsilon, q_0)$ holds. An automaton $\mathcal{A}$ *accepts* an input tree t if there exists a successful run of $\mathcal{A}$ on t. The *language accepted by* $\mathcal{A}$, $L(\mathcal{A})$, is the set of all trees accepted by $\mathcal{A}$. ◇

A run labels each node x either with a pair (v, q) or with #, where the latter indicates that $r(x)$ is not important for the acceptance of the input tree. Please observe that, unlike for NTAs, there is no one-to-one correspondence between the nodes of the tree t and the successful run r: t and r have different arity, and several nodes of the run can refer to the same tree node, whereas other tree nodes are not referenced at all within the run. Moreover, the ordering of successors is important in t, but not in r: the definition of a run only requires the existence of certain successors.

Example 4.12. Let $\mathcal{A} = (\{q_0, \ldots, q_3\}, \{a, b, c\}, q_0, \delta)$ with

$$\begin{aligned} \delta(q_0, a) &= ((0, q_3) \wedge (2, q_1)) \vee (3, q_2) \\ \delta(q_1, b) &= (0, q_0) \wedge (3, q_3) \\ \delta(q_3, a) &= \delta(q_0, b) = \delta(q_3, c) = \mathit{true} \end{aligned}$$

Figure 4.2 shows (the relevant parts of) an input tree t and a successful run r of $\mathcal{A}$ on t. Nodes labelled with # are omitted. Obviously, both $r(3)$ and $r(31)$, which are on different levels of depth within r, refer to node 1, but none refers to node 13.

We will now show that the emptiness problem for alternating tree automata is in ExpTime. Although this result is well-known (Vardi, 1998), to the best of our knowledge there exists no reference containing a detailed description of the decision procedure for the special case of looping automata.[2] We therefore present such a construction in the following.

From Definition 4.11, it becomes clear that in order to test the emptiness for an ATA it is necessary to search for an appropriate input tree t and a successful run r on t, which can have a completely different structure from t. It is possible, however, to test both of them at once: a *strategy tree* is, intuitively, an input tree t whose nodes are labelled with elements of Σ, and where additionally the automaton has left a "footprint" for every state in which (a copy of) the automaton has read the corresponding node.

[2] Vardi (1998) focuses on the constructions required for handling more sophisticated acceptance conditions.

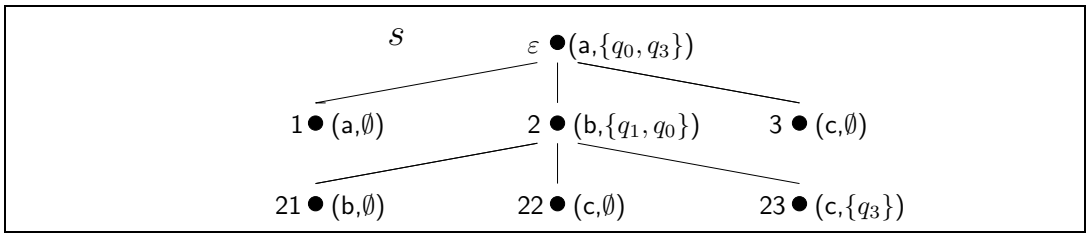

Figure 4.3: A strategy tree for t and r from Example 4.12

Definition 4.13 (Strategy tree). Let $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ be an ATA on k-ary trees. A *strategy tree* for $\mathcal{A}$ is a k-ary tree $s : K^* \to (\Sigma \times 2^Q)$ such that the following holds:

1. for $s(\varepsilon) = (\sigma(\varepsilon), Q(\varepsilon))$, $Q(\varepsilon)$ contains q_0;
2. for every node n with $s(n) = (\sigma(n), Q(n))$ and every $q \in Q(n)$ with $\delta(\sigma(n), q) = \varphi$, there exists a set $S = \{(c_1, q_1), \ldots, (c_m, q_m)\}$ such that
   (a) S satisfies φ,
   (b) for every pair $(c_i, q_i) \in S$, $s(n \cdot c_i)$ is defined and for $s(n \cdot c_i) = (\sigma(i), Q(i))$, $Q(i)$ contains q_i. ◇

Figure 4.3 shows a strategy tree generated from t and r from Figure 4.2. Note that s is a relabelling of t, where the states of $\mathcal{A}$ that visit a node v have been added to v's label. The existence of a strategy tree guarantees the existence of an accepted input tree t and a successful run r on t:

Theorem 4.14. For an ATA $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ on k-ary trees, there exists a strategy tree s iff there exists an input tree t and a successful run r of $\mathcal{A}$ on t.

Proof. For the "if" direction, let t be a k-ary tree which is accepted by $\mathcal{A}$ and r be the corresponding successful run. We construct s incrementally by traversing r breadth-first. The first component of every node n, $\sigma(n)$, is taken directly from $t(n)$. For the second component, we add the appropriate states to $Q(n)$ during the traversal, starting with adding q_0 to $Q(\varepsilon)$, which satisfies the first condition in Definition 4.13. Since r is a successful run, there is a set $S = \{(v_1, q_1), \ldots (v_m, q_m)\}$ that satisfies $\delta(q_0, \sigma(\varepsilon))$ and for that $r(\varepsilon \cdot i) = (\varepsilon \cdot v_i, q_i)$ holds. We therefore add, for every $1 \leq i \leq m$, q_i to $Q(v_i)$. Then, the root node of s satisfies the second condition of Definition 4.13 for the first element q_0 of $Q(\varepsilon)$, and we have added, for every successor of ε in r, the state q_i to the node v_i which $\mathcal{A}$ reads in state q_i.

Now assume we have traversed r up to a node x with $r(x) = (v, q)$. By induction hypothesis, we have added q to $Q(v)$ in the strategy tree while processing the predecessor of x. Again, since r is a run, there is a set S satisfying $\delta(q, t(v))$, thus we can add the corresponding states to the neighbours of v, which ensures that s satisfies Condition 2 for the state q and that the states appearing in $\delta(q, t(v))$ are added to the Q sets of the appropriate neighbours of v. By induction, it follows that this condition is satisfied for every state appearing in a set $Q(v)$.

For the "only-if" direction, we can read read the labelling of an accepted input tree t directly off of the strategy tree s: for every node v, we define $t(v)$ as the first component of $s(v)$. Then we can construct r iteratively, starting with setting $r(\varepsilon) = (\varepsilon, q_0)$, which satisfies the first condition of Definition 4.11. Now let $\varphi = \delta(q_0, t(\varepsilon))$. Since s is a strategy tree, there is a set $S = \{(c_1, q_1), \ldots, (c_m, q_m)\}$ which satisfies φ and for which it holds that $s(\varepsilon \cdot c_i) = (\sigma(i), Q(i))$ with $q_i \in Q(i)$. We then label the first m successors of ε in r with $(c_1, q_1), \ldots, (c_m, q_m)$ and the remaining ones with $\#$, which satisfies the second condition of Definition 4.11 for $r(\varepsilon)$. By construction, $r(i)$ consists of $t(c_i)$ and $\{q_i\}$ for every $1 \leq i \leq m$.

Now assume we have constructed a successful run up to node x with $r(x) = (v, q)$. By induction hypothesis, all predecessors of x satisfy the conditions in Definition 4.11. Since the label of $s(v)$ contains $t(v)$ and q, there is a set $S = \{(c_1, q_1), \ldots, (c_m, q_m)\}$ that satisfies $\delta(q, t(v))$. We can therefore label the first m successors with $(v \cdot c_i, q_i)$, which ensures that also the node x satisfies the conditions for a run. Moreover, since s is a strategy tree, $s(v \cdot c_i)$ contains q_i. Thus it follows by induction that r is a successful run and that t is accepted by $\mathcal{A}$. □

Due to Theorem 4.14, it suffices to test the existence of strategy trees in order to decide the emptiness problem of ATAs, which is also the key to an effective decision procedure. For an ATA $\mathcal{A} = (Q, \Sigma, q_0, \delta)$, the existence of strategy trees can be tested using an NTA $\mathcal{B}$: an alphabet symbol of $\mathcal{B}$ consists of an element of Σ and a subset of Q. In order to deal with transitions to the current node and its father, we have to remember the states contained in the father node's label and the states required for the current label, thus a state of $\mathcal{B}$ is a pair (Q_F, Q_S) of subsets of Q. Intuitively, the first set Q_F consists of the states in the father node's label and the second set Q_S contains the required states (not necessarily all states) for the node itself. Since the root node does not have a predecessor and must contain the state q_0, the initial state of $\mathcal{B}$ is $(\emptyset, \{q_0\})$.

The transition relation of $\mathcal{B}$ ensures that for every state in the node label, the transition function is satisfied by the states contained in the current node, its father, and its children. We therefore take the conjunction of the transition function for all states, transform it into disjunctive normal form and take every disjunct as one possible transition of $\mathcal{B}$'s transition relation. For each of these transitions, which is a conjunction of pairs of children nodes and states, the copy we send to each child contains the corresponding states. This is formalised in the following definition.

Definition 4.15. Let $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ be an ATA. The *strategy automaton* $\mathcal{B} = (P, \Xi, p_0, \Delta)$ is defined as:

- $P = 2^Q \times 2^Q$;
- $\Xi = \Sigma \times 2^Q$;
- $p_0 = (\emptyset, \{q_0\})$;

and the transition relation Δ is defined as follows: for a state $(Q_F, Q_S) \in P$ and a node label (σ, Q_C), if $Q_S \subseteq Q_C$, let $Q_C = \{q_1, \ldots, q_\ell\}$ and $\varphi = \delta(\sigma, q_1) \wedge \ldots \wedge \delta(\sigma, q_\ell)$

and $\varphi' = \varphi_1 \vee \ldots \vee \varphi_m$ for some m be the disjunctive normal form of φ. Let φ'' be the formula φ' where every occurrence of a literal $(0, q)$ is replaced with *true* if $q \in Q_C$ and with *false* otherwise, and every occurrence of $(-1, q)$ is replaced with *true* if $q \in Q_F$ and with *false* otherwise. If $\varphi'' \equiv$ *false*, there are no transitions from the state (Q_F, Q_S) and the letter (σ, Q_C). If $\varphi'' \equiv$ *true*, Δ contains $((Q_F, Q_S), (\sigma, Q_C), (Q_C, \emptyset), \ldots, (Q_C, \emptyset))$.

Otherwise, every disjunct φ_i of φ'' is a conjunction $(c_{i1}, q_{i1}) \wedge \ldots \wedge (c_{in}, q_{in})$ for some n. For every such conjunction, Δ contains the transition $((Q_F, Q_S), (\sigma, Q_C), (Q_C, Q_1), \ldots, (Q_C, Q_k))$, where for $k \in K$, Q_k is defined as $\{q_{ij} \mid q_{ij}$ appears in a literal $(k, q_{ij})\}$.

Additionally, to deal with cases in which Q_k is empty, Δ contains the transitions $((Q_F, \emptyset), (\sigma, Q_C), (\emptyset, \emptyset), \ldots, (\emptyset, \emptyset))$ for all values of Q_F, σ and Q_C. $\diamond$

Since the size of 2^Q is exponential in the size of Q and the length of φ'' can be exponential in the size of φ, both the state space and the transition relation of $\mathcal{B}$ are exponential in the size of the corresponding elements of $\mathcal{A}$. This blow-up cannot be avoided since, as we will see later in this section, it is possible to decide satisfiability for a DL with an ExpTime-complete logic satisfiability problem using an ATA of size polynomial in the size of the input.

We will now show that $\mathcal{B}$ accepts exactly the strategy trees for $\mathcal{A}$. Together with Theorem 4.14, this shows how the emptiness problem for alternating automata can be decided in time exponential in the size of the automaton.

Theorem 4.16. The strategy automaton $\mathcal{B} = (P, \Xi, p_0, \Delta)$ for an ATA $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ accepts an input s iff s is a strategy tree for $\mathcal{A}$.

Proof. For the "if" direction, we will show how to generate a successful run r of $\mathcal{B}$ on s. As s is a strategy tree for $\mathcal{A}$, it holds that $s(\varepsilon) = (\sigma(\varepsilon), Q(\varepsilon))$ with $q_0 \in Q(\varepsilon)$ and for every $q_i \in Q(\varepsilon)$, the PBF $\varphi_i = \delta(q, \sigma(\varepsilon))$ is satisfied by the labels of ε and the nodes $1, \ldots, k$. Then also the conjunction φ_ε of all φ_i is satisfied or equivalently one disjunct of the disjunctive normal form φ_ε. Let that disjunct be $(c_1, q_1) \wedge \ldots \wedge (c_m, q_m)$. If there is a j with $c_j = 0$ then the transition was replaced with *true*, which means that $Q(\varepsilon)$ contains q_j (otherwise, the conjunction would evaluate to *false*). Then, by the construction of Δ, there is a transition $(\emptyset, \{q_0\}), ((\sigma(\varepsilon), Q(\varepsilon)), (Q(\varepsilon), Q_1), \ldots, (Q(\varepsilon), Q_k))$, with every child node $i \in K$ containing all $q \in Q_i$. With $r(\varepsilon) = (\emptyset, \{q_0\})$ and $r(i) = (Q(\varepsilon), Q_i)$ for $i \in K$, we obtain that $r(\varepsilon)$ is labelled with an initial state and there is a transition from ε, and the first component in the label of every child consists of $Q(\varepsilon)$.

Now assume we have constructed r up to a node v with $r(v) = (Q_F, Q_S)$ and $s(v) = (\sigma(v), Q_C)$. Again we obtain that there is a conjunction of transitions $(c_1, q_1) \wedge \ldots \wedge (c_m, q_m)$. If $c_j = -1$ for some j then, since the conjunction evaluates to *true*, we know that $q_j \in Q_F$ holds. By induction hypothesis, this implies that $s(v \cdot (-1))$ contains q_j. The labels of v's children can then be obtained as before.

Conversely, let s be a tree accepted by $\mathcal{B}$, and let r be the corresponding successful run. Then, since $r(\varepsilon)$ is labelled with the initial state $p_0 = (\emptyset, q_0)$, the set Q_ε in $s(\varepsilon) = (\sigma_\varepsilon, Q_\varepsilon)$ contains q_0, which satisfies Condition 1 of Definition 4.13.

Moreover, since r is a successful run, Q_ε is the first component in the label of each of ε's children. Let $(Q_\varepsilon, Q_1), \ldots, (Q_\varepsilon, Q_k)$ be the labels of $r(1), \ldots, r(k)$, and for every $i \in K \cup \{\varepsilon\}$, let the set of states be $Q_i = \{q_{i1}, \ldots, q_{im_i}\}$, where $m_i = \#Q_i$. Then the set of literals $\{(0, q_{\varepsilon 1}), \ldots, (0, q_{\varepsilon m_\varepsilon}), (1, q_{11}), \ldots, (1, q_{1m_1}), (2, q_{21}), \ldots, (k, q_{km_k})\}$ satisfies the conjunction $\delta(q_1, \sigma_\varepsilon) \wedge \delta(q_2, \sigma_\varepsilon) \wedge \ldots \wedge \delta(q_m, \sigma_\varepsilon)$. Since r is a run on s, it holds for every $i \in K$ with $s(i) = (\sigma_i, Q_i^s)$ that $Q_i \subseteq Q_i^s$. Therefore, Condition 2 of Definition 4.13 is satisfied for the root node of s.

Now assume we are at a node v with $s(v) = (\sigma_v, Q_C)$, $r(v) = (Q_F, Q_S)$, and $Q_F = \{q_{F1}, \ldots, q_{Fm_F}\}$. By induction hypothesis, all predecessors of v in s satisfy the conditions for a strategy tree for $\mathcal{A}$, and the father node of v is labelled with (σ_F, Q_F) for some σ_F. Then we obtain Condition 2 of Definition 4.13 in the same way as before, with the only difference that the set of literals additionally contains $(-1, q_{F1}), \ldots, (-1, q_{Fm_F})$. Since each of the states in these transitions is contained in the label of the father node by induction hypothesis, the existence of a transition from $s(v)$ guarantees that v satisfies the conditions of a strategy tree. By induction, this holds for every node of s. □

In the following, we will develop an ExpTime algorithm for the satisfiability problem of $\mathcal{ALC}$ concepts w. r. t. general TBoxes that uses alternating automata. Similar algorithms for different DLs were developed e. g. by Calvanese et al. (1999) and Calvanese, de Giacomo, and Lenzerini (2002).

4.4.1 An ATA Algorithm for $\mathcal{ALC}$ with General TBoxes

Employing ATAs for the satisfiability test allows for a procedure that is more intuitive and "tableau-like" than the NTA algorithm in Section 4.3.1. Since the automaton can remain in the same node, we do not have to restrict the node labels to Hintikka sets; instead, we can handle conjunctions and disjunctions with transitions. We can also drop the restriction that the n-th child always stands for the n-th existential restriction; instead, we can simply say that *some* child satisfies the corresponding concept. Finally, a single child can be both an r- and an s-successor for two different roles r and s.[3]

In order to deal with this additional freedom without using labelled edges, we require additional alphabet symbols: for a role name r, the symbol s_r in a node label means that the node is an r-successor. To improve readability, we define, for a set of role names $\mathsf{N_R}$ and a TBox $\mathcal{T}$, the following shortcuts:

$$RC = \{\mathsf{s}_\mathsf{r} \mid \mathsf{r} \in \mathsf{N_R}\} \quad \text{and} \quad \mathsf{C}_\mathcal{T} = \prod_{\mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}} (\dot{\neg}\mathsf{D} \sqcup \mathsf{E})$$

Thus, for an input $\mathsf{C}, \mathcal{T}$, the node labels contain concept names and role symbols. The arity k is, as in the case of NTA, the number of existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$. Nodes that are not needed for an existential restriction are labelled with the dummy

[3] This feature is not necessary for $\mathcal{ALC}$, but it is included here in order to maintain consistency with the $\mathcal{ALCIO}$ algorithm in Section 5.2.2.

$$
\begin{aligned}
\delta(\mathit{START}, \sigma) &= (0, \mathsf{C}) \wedge (0, \mathit{GCI}) \\
\delta(\mathit{GCI}, \sigma) &= (0, \mathsf{C}_{\mathcal{T}}) \wedge \bigwedge_{i=1}^{k} ((i, \mathit{GCI}) \vee (i, \#)) \\
\delta(\mathsf{s}_\mathsf{r}, \sigma) &= \begin{cases} \mathit{true}, & \text{if } \mathsf{s}_\mathsf{r} \in \sigma \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\delta(\overline{\mathsf{s}}_\mathsf{r}, \sigma) &= \begin{cases} \mathit{true}, & \text{if } \mathsf{s}_\mathsf{r} \notin \sigma \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\delta(\mathsf{D}, \sigma) &= \begin{cases} \mathit{true}, & \text{if } \mathsf{D} \in \sigma \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\delta(\neg\mathsf{D}, \sigma) &= \begin{cases} \mathit{true}, & \text{if } \mathsf{D} \notin \sigma \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\delta(\mathsf{D} \sqcap \mathsf{E}, \sigma) &= (0, \mathsf{D}) \wedge (0, \mathsf{E}) \\
\delta(\mathsf{D} \sqcup \mathsf{E}, \sigma) &= (0, \mathsf{D}) \vee (0, \mathsf{E}) \\
\delta(\exists \mathsf{r}.\mathsf{D}, \sigma) &= \bigvee_{i=1}^{k} ((i, \mathsf{s}_\mathsf{r}) \wedge (i, \mathsf{D})) \\
\delta(\forall \mathsf{r}.\mathsf{D}, \sigma) &= \bigwedge_{i=1}^{k} ((i, \overline{\mathsf{s}}_\mathsf{r}) \vee (i, \mathsf{D}) \vee (i, \#)) \\
\delta(\#, \sigma) &= \begin{cases} \mathit{true}, & \text{if } \sigma = \# \\ \mathit{false}, & \text{otherwise} \end{cases} \\
\delta(q, \#) &= \begin{cases} \mathit{true}, & \text{if } q = \# \\ \mathit{false}, & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.4: $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ transition relation

symbol #. The $\overline{\mathsf{s}}_\mathsf{r}$ states are used to ensure that s_r is *not* contained in a node label, i. e. that the corresponding node is not an r-successor.

The following construction of an automata algorithm for $\mathcal{ALC}$ is based on an algorithm for the hybrid μ-calculus by Sattler and Vardi (2001).

Definition 4.17 (ATA for C w. r. t. $\mathcal{T}$). Let C be an $\mathcal{ALC}$ concept and $\mathcal{T}$ be a TBox in negation normal form, with k being the number of existential restrictions in $\mathit{sub}(\mathsf{C}) \cup \mathit{sub}(\mathsf{C}_{\mathcal{T}})$. Moreover, let $\mathsf{N}_\mathsf{C}(\mathsf{C}, \mathcal{T})$ be the set of concept names occurring in C or $\mathcal{T}$, and let $RC(\mathsf{C}, \mathcal{T}) = \{\mathsf{s}_\mathsf{r} \mid \mathsf{r} \text{ occurs in } \mathsf{C} \text{ or } \mathcal{T}\}$. Then the *ATA for* C *w. r. t.* $\mathcal{T}$ is defined as $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, \Sigma, q_0, \delta)$ with

- $Q = \mathit{sub}(\mathsf{C}) \cup \mathit{sub}(\mathsf{C}_{\mathcal{T}}) \cup RC(\mathsf{C}, \mathcal{T}) \cup \{\overline{\mathsf{s}}_\mathsf{r} \mid \mathsf{s}_\mathsf{r} \in RC(\mathsf{C}, \mathcal{T})\} \cup \{\mathit{START}, \mathit{GCI}, \#\}$;
- $\Sigma = \{\sigma \mid \sigma \subseteq \mathsf{N}_\mathsf{C}(\mathsf{C}, \mathcal{T}) \cup RC(\mathsf{C}, \mathcal{T})\} \cup \{\#\}$;
- $q_0 = \mathit{START}$;

- for $q \in Q$ and $\sigma \in \Sigma$, δ is defined as in Figure 4.4. $\diamond$

Observe that the size of the automaton, i. e. the number of its states, is polynomial in the size of C and $\mathcal{T}$. This translation is more straightforward than in the case of an NTA in the sense that for a concept name D, we simply have to check if D is contained in the node label; conjunctions and disjunctions translate into conjunctions and disjunctions in the transition function, and the transitions for an existential restriction $\exists \mathsf{r}.\mathsf{C}$ or value restriction $\forall \mathsf{r}.\mathsf{C}$ ensure that there is a successor labelled with s_r and C or that all non-dummy successors which are labelled with s_r are also labelled with C, respectively. TBoxes are handled by enforcing that every node satisfying an existential restriction also satisfies all GCIs. The transitions bear a strong resemblance with the tableau rules in Figure 3.2: there is one type of transitions for every constructor and an additional one for GCIs. The main difference is that, since an automaton does not generate, but recognise models, it sends a copy of itself to a node v in order to make sure that $t(v)$ contains a concept C, rather than adding C to $t(v)$. For existential restrictions, this involves testing every successor as a possible candidate. Additionally, we require the transitions for dummy nodes because the automaton operates on full k-ary trees.

Theorem 4.18. For an $\mathcal{ALC}$ concept C and TBox $\mathcal{T}$, the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ accepts a non-empty language iff C is satisfiable w. r. t. $\mathcal{T}$.

Proof. For the "if" direction, let $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ be a model for C w. r. t. $\mathcal{T}$, and let k be as in Definition 4.17. As $\mathcal{I}$ is a model, there is a $d_\varepsilon \in \Delta^\mathcal{I}$ with $d_\varepsilon \in \mathsf{C}^\mathcal{I}$. We construct a strategy tree s for $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ by setting

$$s(\varepsilon) = (\{\mathsf{D} \in \mathsf{N_C} \mid d_\varepsilon \in \mathsf{D}^\mathcal{I}\}, \{\mathsf{D} \in \mathit{sub}(\mathsf{C},\mathcal{T}) \mid d_\varepsilon \in \mathsf{D}^\mathcal{I}\} \cup \{\overline{\mathsf{s}}_\mathsf{r} \mid \mathsf{r} \in \mathsf{N_R}\}),$$

i. e. the first component (σ_ε) contains all concept names to which d_ε belongs, and the second one (Q_ε) contains all subconcepts of C and $\mathcal{T}$ to which d_ε belongs. Then, for a node v that is already labelled using an individual d, we label v's children nodes as follows: for every concept $\exists \mathsf{r}.\mathsf{D} \in Q_v$, we choose an individual $d_i \in \Delta^\mathcal{I}$ with $(d, d_i) \in \mathsf{r}^\mathcal{I}$ and $d_i \in \mathsf{D}^\mathcal{I}$ (such an individual exists because $\mathcal{I}$ is a model), and label the child node $v \cdot i$ in the way described above—with the only difference that, if $(d, d_i) \in \mathsf{r}^\mathcal{I}$ holds, then σ_i and Q_i additionally contain s_r, and Q_i does not contain $\overline{\mathsf{s}}_\mathsf{r}$. If the number of existential restrictions in Q_ε is less than k, we label the remaining children with $(\#, \#)$.

It follows by induction over the length of the concepts in $\mathit{sub}(\mathsf{C},\mathcal{T})$ that, for every node v and every $q \in Q_v$, the transition function evaluates to *true*: for concept names, negated concept names, role symbols s_r, and dummy nodes, this follows directly from the construction of s. For conjunctions and disjunctions, it follows by structural induction, and for existential restrictions, it again follows by construction that one of the children nodes is labelled with the appropriate concept and role. For value restrictions, assume there is a node v labelled with the concepts for an individual d such that $\forall \mathsf{r}.\mathsf{D} \in Q_v$, and that there is an i such that $Q_{v \cdot i} \cap \{\overline{\mathsf{s}}_\mathsf{r}, \mathsf{D}, \#\} = \emptyset$. Then there is an individual $d_i \in \Delta^\mathcal{I}$ such that $(d, d_i) \in \mathsf{r}^\mathcal{I}$ and $d_i \notin \mathsf{D}^\mathcal{I}$. This contradicts

the assumption that $d \in (\forall \mathsf{r}.\mathsf{D})^{\mathcal{I}}$. Finally, since $\mathcal{I}$ is a model, every individual satisfies every GCI, and thus $\mathsf{C}_{\mathcal{T}}$ is in the label of every non-dummy node and, since $d_\varepsilon \in \mathsf{C}^{\mathcal{I}}$, Q_ε contains C.

Conversely, let s be a k-ary strategy tree for $\mathcal{A}_{\mathsf{C},\mathcal{T}}$. For a node $v \in K^*$ with $s(v) = (\sigma, Q)$ we will refer to σ and Q by $\sigma(v)$ and $Q(v)$. We define a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for C w. r. t. $\mathcal{T}$ by setting $\Delta^{\mathcal{I}} = \{v \in K^* \mid \mathsf{C}_{\mathcal{T}} \in \sigma(v)\}$, for a concept name A, $\mathsf{A}^{\mathcal{I}}$ is defined as $\{v \in \Delta^{\mathcal{I}} \mid \mathsf{A} \in \sigma(v)\}$, and for a role name r, $\mathsf{r}^{\mathcal{I}}$ is defined as $\{(v, w) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid \mathsf{s}_\mathsf{r} \in \sigma(w) \text{ and } w = v \cdot i \text{ for some } i \in K\}$. We will show by induction over the structure of the elements of Q that the individual v satisfies all concepts contained in $Q(v)$. For concept names and their negations, this follows directly from the definition of δ and the fact that s is a strategy tree; for conjunctions and disjunctions, a similar argument holds. For an existential restriction $\exists \mathsf{r}.\mathsf{D} \in \sigma(v)$, there exists an i such that $Q(v \cdot i)$ contains s_r, D and $\mathsf{C}_{\mathcal{T}}$. By the definition of $\mathcal{I}$, $\Delta^{\mathcal{I}}$ contains $v \cdot i$ and $\mathsf{r}^{\mathcal{I}}$ contains $(v, v \cdot i)$; and by induction, the individual $v \cdot i$ satisfies D.

For a value restriction $\forall \mathsf{r}.\mathsf{D} \in \sigma(v)$ and every $i \in K$, $Q(v \cdot i)$ contains $\overline{\mathsf{s}}_\mathsf{r}$, D, or $\#$. In the first case, $\mathsf{s}_\mathsf{r} \notin \sigma(v \cdot i)$ holds and thus by construction also $(v, v \cdot i) \notin \mathsf{r}^{\mathcal{I}}$. In the third case, $\sigma(v \cdot i) = \#$ holds and therefore $v \cdot i$ is not contained in $\Delta^{\mathcal{I}}$. In the second case, $\mathsf{D} \in \sigma(v \cdot i)$ holds and $v \cdot i \in \mathsf{D}$ follows by induction hypothesis. All elements of $\Delta^{\mathcal{I}}$ satisfy $\mathsf{C}_{\mathcal{T}}$ and thus all GCIs, and since $Q(\varepsilon)$ contains both $\mathsf{nnf}(\mathsf{C})$ and $\mathsf{C}_{\mathcal{T}}$, ε is contained in $\Delta^{\mathcal{I}}$ and satisfies C. □

From Theorem 4.18, we again obtain the ExpTime complexity result for $\mathcal{ALC}$ with GCIs since the size of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is polynomial in the size of C and $\mathcal{T}$ and the emptiness test for $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is exponential in the size of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$.

Corollary 4.19. The satisfiability problem for $\mathcal{ALC}$ concepts w. r. t. general TBoxes is in ExpTime.

The advantage of ATAs over NTAs is the more intuitive construction: in the ATA algorithm, all language constructors are dealt with by the transition function whereas, in the NTA algorithm, the propositional constructors and clashes are handled by the definition of Hintikka sets, GCIs are handled by the definition of $\mathsf{C}, \mathcal{T}$-compatibility, and only existential and value restrictions are handled by the transition relation. Moreover, the handling of value and existential restriction is merged in the NTA, where every r-successor required by an existential restriction is required to satisfy all value restrictions concerning r, whereas in the ATA, existential and value restrictions are tested independently. In the words of Muller and Schupp (1995), "alternating automata really provide the 'natural model' for automata working on infinite inputs". Vardi (1997) argues that alternating automata "enable one to decouple the logic from the algorithmics", i. e. they allow the user to focus on the logical requirements of the language under consideration and leave the combinatorial explosion of the state space in the step that is handled automatically.

More specifically, for the application in the area of DLs, Calvanese et al. (2002) note that ATA algorithms are *intuitive* like tableau rules, *modular* since they handle each construct separately, *short* since the encoding is polynomial, and *computationally adequate* since they are worst-case optimal for ExpTime-complete logics.

Having described tableau and automata algorithms deciding satisfiability in $\mathcal{ALC}$ in detail, we can take a closer look at their differences and the influence of these differences on efficiency in theory and practice. The emptiness test for the NTA from Definition 4.2 (and thus also the one for the NTA testing the existence of a strategy tree for an ATA) goes in *bottom-up* direction: it starts with the set of all possible node labels (i. e. Hintikka sets for the input) and removes those from which there is no transition, e. g. sets containing $\exists \mathsf{r}.\mathsf{A}$ and $\forall \mathsf{r}.\neg\mathsf{A}$. In the next iteration, all sets are removed that require a transition to one of the previously removed sets, e. g. sets containing $\exists \mathsf{s}.(\exists \mathsf{r}.\mathsf{A} \sqcap \forall \mathsf{r}.\neg\mathsf{A})$. If we continue this iteration until no further states are removed and we can still find a set containing the input, then we know that there exists a successful run (although the automata approach does not provide an accepted input tree or a successful run as a witness for satisfiability).

Since this algorithm works bottom-up, it is deterministic in spite of the non-deterministic transition relation: when testing the satisfiability of a set containing $\mathsf{C} \sqcup \mathsf{D}$, it is only necessary to test for the existence of sets containing C or D, which requires time linear in the size of the automaton. However, this procedure obviously only works if it starts with *all* possible sets, and is therefore best-case exponential. As mentioned in Example 4.7, it is possible to perform the emptiness test in top-down direction, thus avoiding the generation of unnecessary states, but this requires measures to ensure termination, and it leads to a non-deterministic procedure, which does not provide an ExpTime complexity bound anymore. In Chapter 6, we show how a non-deterministic top-down test can be useful to obtain PSpace results for certain logics.

In contrast, the tableau algorithm works in a *top-down* manner: it starts by labelling a node with the initial concept and tries to generate successor nodes that do not contain a clash, e. g. an r-successor labelled with C in order to satisfy an existential restriction $\exists \mathsf{r}.\mathsf{C}$. This procedure is goal-directed because every node that is created and every concept that is added to a node label is relevant for the satisfiability of the input. However, like the top-down automata test sketched above, this requires special measures in order to ensure termination, and it leads to a non-deterministic procedure in the presence of non-deterministic rules, e. g. the $\sqcup$-rule or the $\mathcal{T}$-rule. The similarities between the rules and the definition of the transition relation raise the question whether it is possible to automatically generate an ExpTime automata algorithm from a tableau algorithm. In Chapter 7, we address this issue.

One can also ask if the relation between tableau and automata algorithms is so close that the advantages of TAs, in particular the performance improvement by optimisations, can be transferred to AAs simply by translating an automaton into an input for a tableau algorithm. This goal is pursued in the next chapter.

Chapter 5

Translation of Alternating Automata into Description Logics

In this chapter, we address the question if the good performance of tableau algorithms in practice can be transferred automatically to automata algorithms through translation. To this end, we develop ways to translate alternating one-way and two-way automata (see Section 4.4) into TBoxes in comparably inexpressive DLs, so that the emptiness problem of the automaton can be reduced to satisfiability of a certain concept w. r. t. the corresponding TBox. With this approach, we can use the existing implementations of tableau algorithms to perform the emptiness test, and check if the optimisations of the reasoners are able to achieve an acceptable calculation time.

In Section 5.1 we show how one-way alternating automata can be translated into the inexpressive DL $\mathcal{ELU}_f$, from which we obtain that in this logic, concept satisfiability w. r. t. general TBoxes is ExpTime-hard. This translation is extended in Section 5.2 to two-way automata and the more expressive DL $\mathcal{FLEUI}_f$. In Section 5.2.1, we define an ATA algorithm for the logic $\mathcal{ALCIO}$, which, together with the above-mentioned translation, enables us to use the existing $\mathcal{SHIQ}$ (Horrocks et al., 2000a) implementations FaCT (Horrocks, 1998b) and Racer (Haarslev and Möller, 2001b) to reason about nominals although the $\mathcal{SHIQ}$ language does not provide this constructor. Section 5.2.3 describes the empirical results obtained with this translation.

This chapter is based on work that was previously published by Hladik and Sattler (2003); and Hladik (2003).

5.1 Translation of One-Way Automata into $\mathcal{ELU}_f$

The automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ in Definition 4.17 does not really exploit the possibilities of *two*-way automata because it does not make use of (-1)-transitions leading to the father node. This simplifies the translation of the automaton into a TBox significantly and only requires an inexpressive DL. We will therefore call automata which use only

transitions (c, q) with $q \in Q$ and $c \in K_0 := \{0, \ldots, k\}$ *one*-way alternating automata and begin with a translation of one-way automata into the DL $\mathcal{ELU}_f$, which is formally defined below.

Definition 5.1 ($\mathcal{ELU}_f$). Let $\mathsf{N_C}$ be a set of concept names and $\mathsf{N_F}$ a set of feature names. The set of $\mathcal{ELU}_f$ *concepts* over $\mathsf{N_C}$ and $\mathsf{N_F}$ is inductively defined as follows:

- $\top$, $\bot$, and all concept names $\mathsf{C} \in \mathsf{N_C}$ are concepts;
- if C and D are concepts, then $\mathsf{C} \sqcup \mathsf{D}$ and $\mathsf{C} \sqcap \mathsf{D}$ are concepts;
- if C is a concept and $\mathsf{f} \in \mathsf{N_F}$ is a feature name, then $\exists \mathsf{f}.\mathsf{C}$ is a concept.

The semantics of these constructors is given in Figure 2.1 on Page 28. GCIs and TBoxes are defined as for $\mathcal{EL}$ (Definition 2.2). ◇

In the presence of GCIs, the constructors provided by $\mathcal{ELU}_f$ are sufficiently expressive to translate the transition relation of an alternating *one*-way automaton $\mathcal{A}$ into a general TBox $\mathit{tr}(\mathcal{A})$ in such a way that the concept Q_0, which represents the initial state of the automaton, is satisfiable w. r. t. $\mathit{tr}(\mathcal{A})$ iff $\mathcal{A}$ accepts a non-empty language. The idea is that, from a model $\mathcal{I}_\mathcal{A}$ for Q_0 w. r. t. $\mathit{tr}(\mathcal{A})$, it is possible to construct a tree t accepted by $\mathcal{A}$ and a corresponding successful run r. For this purpose, both alphabet symbols and states are represented with concept names. An individual of $\mathcal{I}_\mathcal{A}$ represents a node in t and thus belongs to the interpretation $\mathsf{A}^\mathcal{I}$ of exactly one alphabet symbol a. Moreover, it belongs to the interpretation $\mathsf{Q}_i^\mathcal{I}$ for every state q_i in which the automaton operates on the corresponding node. The transition relation of $\mathcal{A}$ is translated into a set of GCIs which ensure that, for every pair (A, Q) that the individual belongs to, the corresponding neighbour nodes contain the required states. The structure of this tree is represented by roles: f_i stands for the relation between the father node and its i-th child node. In order to ensure that there is a unique individual for each child, these roles have to be features.

Definition 5.2 (Translation tr into $\mathcal{ELU}_f$). Let $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ be an alternating one-way automaton with $Q = \{q_0, \ldots, q_{i_{\max}}\}$ and $\Sigma = \{\sigma_0, \ldots, \sigma_{j_{\max}}\}$. The translation of $\mathcal{A}$ into an $\mathcal{ELU}_f$ TBox $\mathit{tr}(\mathcal{A})$ is defined as follows: for each $q_i \in Q$ we use a concept name Q_i, for each $\sigma_j \in \Sigma$, we use a concept name A_j, and we set

$$
\begin{array}{lcl}
\mathit{tr}(\mathcal{A}) & := & \{\mathsf{G}\top, \mathsf{G}\bot\} \cup \bigcup\limits_{q \in Q, \sigma \in \Sigma} \mathit{tr}(\delta(q, \sigma)), \text{ where} \\
\mathsf{G}\top & := & \top \sqsubseteq \mathsf{A}_1 \sqcup \mathsf{A}_2 \sqcup \ldots \sqcup \mathsf{A}_{j_{\max}}, \\
\mathsf{G}\bot & := & \bigsqcup\limits_{0 \le i < j \le j_{\max}} (\mathsf{A}_i \sqcap \mathsf{A}_j) \sqsubseteq \bot, \\
\mathit{tr}(\delta(q, \sigma)) & := & \mathit{tr}(q) \sqcap \mathit{tr}(\sigma) \sqsubseteq \mathit{tr}(\varphi) \quad \text{if } \delta(q, \sigma) = \varphi,
\end{array}
$$

and the translation of φ, q, and σ is defined as follows:

$$
\begin{array}{llllll}
\mathit{tr}(q_i) & := & \mathsf{Q}_i \quad \text{for } q_i \in Q, & \mathit{tr}(\sigma_i) & := & \mathsf{A}_i \quad \text{for } \sigma_i \in \Sigma, \\
\mathit{tr}(\alpha \wedge \beta) & := & \mathit{tr}(\alpha) \sqcap \mathit{tr}(\beta), & \mathit{tr}(\alpha \vee \beta) & := & \mathit{tr}(\alpha) \sqcup \mathit{tr}(\beta), \\
\mathit{tr}(\mathit{true}) & := & \top, & \mathit{tr}(\mathit{false}) & := & \bot, \\
\mathit{tr}(0, q) & := & \mathit{tr}(q), & \mathit{tr}(i, q) & := & \exists \mathsf{f}_i.\mathit{tr}(q) \text{ for } 0 < i \le k.
\end{array}
$$

◇

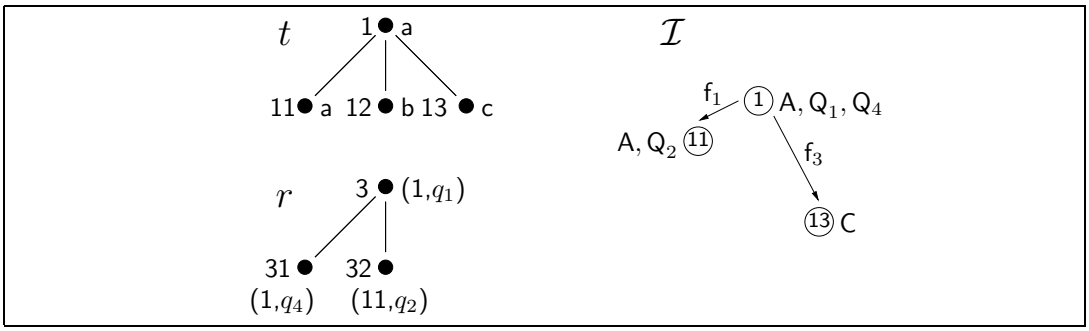

Figure 5.1: Translation of a tree and a successful run into a model

We will see that tr ensures that each model $\mathcal{I}$ of $\mathsf{tr}(\mathcal{A})$ corresponds to a successful run r on some tree t. Firstly, a node x in r is labelled with a node v in t which, in turn, is labelled with exactly one $\sigma \in \Sigma$. Thus each x in r is associated with one $\sigma \in \Sigma$. To express this fact in $\mathsf{tr}(\mathcal{A})$, we use the extra GCIs $\mathsf{G}\top$ and $\mathsf{G}\bot$: they guarantee that every individual of $\mathcal{I}$ is an instance of exactly one $\mathsf{tr}(\sigma_i)$.[1]

Next, it will turn out to be useful to have the inverse tr^{-1} of tr, which is possible since tr is "almost" injective: the only ambiguity concerns q and $(0, q)$ since they are both mapped to Q by the function tr. However, this ambiguity can easily be resolved by agreeing to set $\mathsf{tr}^{-1}(Q)$ to $(0, q)$ if Q appears on the right hand side of a GCI and to q otherwise.

Example 5.3. Figure 5.1 shows an example for the translation of a tree and a successful run into a model for a single node. The automaton is in state q_1 and reads node 1 which is labelled with a. For $\delta(a, q_1) = (1, q_1) \vee ((1, q_2) \wedge (0, q_4))$, the transition function yields the GCI $\mathsf{A} \sqcap \mathsf{Q}_1 \sqsubseteq \exists \mathsf{f}_1.\mathsf{Q}_1 \sqcup ((\exists \mathsf{f}_1.\mathsf{Q}_2) \sqcap \mathsf{Q}_4)$. In our example, the transition function is satisfied via the second disjunct, and thus the individual 1 in the model $\mathcal{I}$ is an instance of both Q_1 and Q_4. Observe that a model $\mathcal{I}$ of $\mathsf{tr}(\mathcal{A})$ might have a structure different from

- an input tree t since nodes in $\mathcal{I}$ might have no f_i-successor for some i and $\mathcal{I}$ might not be a tree; and
- a successful run r since different nodes of r that refer to the same node in t are represented by the same individual in $\mathcal{I}$: intuitively, we label an individual $d \in \Delta^{\mathcal{I}}$ with the concept A of $t(d)$ and "collect" all Q_i concepts from nodes in r that refer to $t(d)$. Thus, some individuals are instances of several Q_i concepts, like 1, while others are instances of none, like 13.

Lemma 5.4. The language accepted by an alternating automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ is non-empty iff $\mathsf{tr}(q_0)$ its satisfiable w. r. t. $\mathsf{tr}(\mathcal{A})$.

[1] Since each node is labelled with exactly one alphabet symbol, it is also possible to translate the alphabet symbols using a binary coding mechanism which requires only $\log_2 n$ concept names for n alphabet symbols. However, this would not reduce the number of GCIs in the TBox, which would still be exponential in n. Hence, we stick with the linear translation, which also improves readability.

Proof. We start with the "only-if"-direction. For this purpose, we fix some notation: we use L (for "literal") to denote the set of concepts appearing on the right-hand side of GCIs in $\mathit{tr}(\mathcal{A})$,

$$L := \{\exists \mathsf{f}_i.\mathsf{Q}_j \mid i \in K_0, 0 \leq j \leq i_{\max}\} \cup \{\mathsf{Q}_j \mid 0 \leq j \leq i_{\max}\},$$

and we use $\mathcal{B}^+(L)$ for the set of positive Boolean concepts analogous to positive Boolean formulas in Definition 4.10, with the symbols $\wedge, \vee, \mathit{true}$, and false replaced with $\sqcap, \sqcup, \top$, and $\bot$, respectively.

Let $\mathcal{T} = \mathit{tr}(\mathcal{A})$, $\mathsf{Q}_0 = \mathit{tr}(q_0)$, and let t be contained in $\mathcal{L}(\mathcal{A})$ where r is a successful run of $\mathcal{A}$ on t. We construct a model $\mathcal{I}$ of Q_0 w. r. t. $\mathcal{T}$ as follows:

$$\begin{array}{rcll} \Delta^{\mathcal{I}} & := & \{v \in K^* \mid t(v) \neq \#\}, & \\ \mathsf{f}_c^{\mathcal{I}} & := & \{(\ell, \ell \cdot c) \mid \{\ell, \ell \cdot c\} \subseteq \Delta^{\mathcal{I}}\} & \text{for every } c \in K, \\ \mathsf{A}_i^{\mathcal{I}} & := & \{v \mid t(v) = \sigma_i\}, & \text{for every } \sigma_i \in \Sigma, \\ \mathsf{Q}_i^{\mathcal{I}} & := & \{v \mid \text{ there is an } x \text{ in } r \text{ with } r(x) = (v, q_i)\}, & \text{for every } q_i \in Q. \end{array}$$

In order to prove that $\mathcal{I}$ is a model of Q_0 w. r. t. $\mathcal{T}$, we show that

1. $\mathsf{Q}_0^{\mathcal{I}} \neq \emptyset$ holds and
2. each individual t of $\mathcal{I}$ satisfies each GCI in $\mathcal{T}$.

Now (1) holds by definition of $\mathcal{I}$ since the $r(\varepsilon) = (\varepsilon, q_0)$ and thus $\varepsilon \in \mathsf{Q}_0^{\mathcal{I}}$. For (2), we distinguish three classes of GCIs in $\mathcal{T}$:

1. GCIs $\mathsf{G} \top$ and $\mathsf{G} \bot$,
2. GCIs of the form $\mathsf{Q} \sqcap \mathsf{A} \sqsubseteq \bot$ resulting from the translation of transitions $\delta(q, \sigma) = \mathit{false}$, and
3. GCIs $\mathsf{Q} \sqcap \mathsf{A} \sqsubseteq \mathsf{C}$, for some concept $\mathsf{C} \in \mathcal{B}^+(L) \setminus \{\bot\}$.

For the first class, $\mathcal{I}$ satisfies $\mathsf{G} \top$ and $\mathsf{G} \bot$ by definition since every node v in t is labelled with exactly one letter σ. For the remainder, consider a GCI $\mathsf{Q}_i \sqcap \mathsf{A}_j \sqsubseteq \mathsf{C}$ in $\mathcal{T}$ with pre-image $\delta(q_i, \sigma_j) = \varphi$ and some $v \in \mathsf{Q}_i^{\mathcal{I}} \cap \mathsf{A}_j^{\mathcal{I}}$. By definition of $\mathcal{I}$, there exists a node x with $r(x) = (v, q_i)$ and $t(v) = \sigma_j$. For the second class, if $\varphi = \mathit{false}$, then $\mathsf{C} = \bot$, and the existence of v is a contradiction to r being a run. For the third class, the definition of a run implies the existence of a set $S = \{i_1, \ldots, i_m\}$ and functions a, c, and s such that

- $\{(a(i), q_{c(i)}) \mid i \in S\}$ satisfies φ and
- there are successors $x \cdot s(i_1), \ldots, x \cdot s(i_s)$ of x which are labelled with the corresponding pairs, i. e. $r(x \cdot s(j)) = (v \cdot a(j), q_{c(j)})$ holds for all $j \in S$.

By construction of $\mathcal{I}$, there exist $\mathsf{f}_{t(j)}$-successors u_j of v with $u_j \in \mathsf{Q}_{c(j)}^{\mathcal{I}} \sqcap \mathsf{A}^{\mathcal{I}}$ for $\mathsf{A} := \mathit{tr}(t(v \cdot a(j)))$. This ensures that $v \in \mathsf{C}^{\mathcal{I}}$, which concludes the proof of the "only-if"-direction.

For the "if"-direction, we will show how to construct a tree t and a successful run r of $\mathcal{A}$ on t from a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of Q_0 w. r. t. $\mathcal{T}$. We define the auxiliary functions

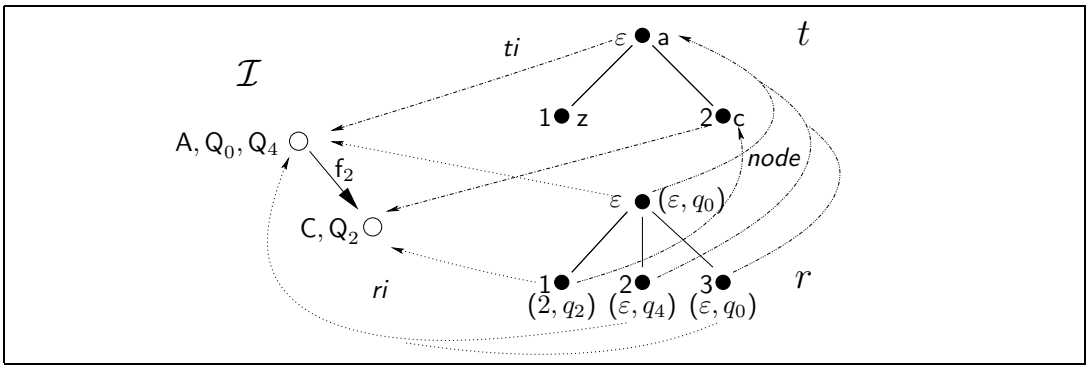

Figure 5.2: Translation of a model into a tree and a successful run

- *ti* and *ri*, which map nodes of t and r, respectively, to individuals of $\Delta^{\mathcal{I}}$,
- *node* and *state*, which map a node of r to the node and state component of its label, i. e. if $r(x) = (v, q)$, then $\mathit{node}(x) = v$ and $\mathit{state}(x) = q$, and
- $\mathit{letter} : \Delta^{\mathcal{I}} \to \{\mathsf{A}_i \mid 1 \leq i \leq j_{\max}\}$ and $\mathit{states} : \Delta^{\mathcal{I}} \to 2^{\{\mathsf{Q}_i \mid 1 \leq i \leq i_{\max}\}}$, assigning, to each individual, the unique concept A_x and the set of Q_i concepts it is an instance of (*letter* is well-defined since $\mathcal{I}$ is a model of $\mathsf{G}\top$ and $\mathsf{G}\bot$).

Example 5.5. In Figure 5.2, we show a model $\mathcal{I}$ together with a tree t, a successful run r, and some of the auxiliary functions. The tree node 1 is assumed to be related to a dummy individual # and its letter $z = \mathit{letter}(\#)$, and only some nodes of r are presented. In the run r, the nodes ε, 2, and 3 are labelled with the same tree node ε since $\mathit{ri}(\varepsilon)$ is an instance of Q_0 and Q_4, and we might need nodes 2 and 3 for a run involving transitions $\delta(a, q_0) = (0, q_4) \wedge (0, q_0) \wedge \ldots$

Intuitively, t is an an unravelling of $\mathcal{I}$ (which need not be tree-shaped), and r is an unravelling with "duplicate" successors (in case several copies of the automaton operate on the same node). Formally, t and r are defined as follows. We begin the construction of t with an individual $d_\varepsilon \in \mathsf{Q}_0^{\mathcal{I}}$. Such an individual exists since $\mathcal{I}$ is a model of Q_0 w. r. t. $\mathcal{T}$. Moreover, we fix some dummy individual $\# \in \Delta^{\mathcal{I}}$. We define $t(\varepsilon) := \mathit{tr}^{-1}(\mathit{letter}(d_\varepsilon))$, i. e. the root node of the tree is labelled with the alphabet symbol in the label of d_ε, and $\mathit{ti}(\varepsilon) := d_\varepsilon$, i. e. the root node is associated with d_ε. Then, for each v such that $t(v)$ is already defined and for each $c \in K$, we do the following:

- if $\mathit{ti}(v)$ has an f_c-successor d_c, define $\mathit{ti}(v \cdot c) := d_c$ and $t(v \cdot c) := \mathit{tr}^{-1}(\mathit{letter}(d_c))$, i. e. we extend the labelling of t to cover the new individual (this is well-defined since f_c is functional);
- otherwise, define $\mathit{ti}(v \cdot c) := \#$ and $t(v \cdot c) := \mathit{tr}^{-1}(\mathit{letter}(\#))$, i. e. label the node as a dummy.

After the input tree t, we now define a successful run r of $\mathcal{A}$ on t as follows. Firstly, set $r(\varepsilon) := (\varepsilon, q_0)$ and $\mathit{ri}(\varepsilon) := d_\varepsilon$. Secondly, if $\mathit{ri}(x) = d$ is already fixed, we define

$d_0 := d$ and d_i as the f_i-successor of d, if there exists one, and $d_i := \#$ otherwise. Then we fix, for every d_i and every $q_{i_j} \in \mathit{states}(d_i)$, a different successor $x \cdot i_j$ of x with $r(x \cdot i_j) = (\mathit{node}(x) \cdot i, q_{i_j})$ and $\mathit{ri}(x \cdot i_j) = d_i$. Finally, if $r(x \cdot c)$ is not fixed through the previous step, we set $r(x') = \#$ for each node x' in the sub-tree below $x \cdot c$ including $x \cdot c$. Thus, for every concept Q_i that d or a successor of d is an instance of, there is a successor of d in r labelled with q_i and the corresponding node in t.

To prove that r is a run on t, we first prove that the ri and ti functions are defined properly, i. e. if $\mathit{node}(x) = v$ for a node v in the tree and a node x in the run, then both v and x refer to the same individual in $\Delta^{\mathcal{I}}$.

Claim: For all nodes x of r, if $r(x) \neq \#$, then $\mathit{ri}(x) = \mathit{ti}(\mathit{node}(x))$.

Proof of the claim. The proof is by induction on the depth of nodes in r. For the root node ε of r, the claim holds by definition. Now let x be a node of r with $r(x) \neq \#$ for which the claim holds. Let $r(x) = d$ and consider a successor $x \cdot c$ of x. If $r(x \cdot c) \neq \#$, then there are i, j such that d has an f_i-successor $d_i \in \mathsf{Q}_j^{\mathcal{I}}$, or $\mathit{ri}(x \cdot c) = d$, which means that $d \in \mathsf{Q}_j^{\mathcal{I}}$ and $i = 0$. Then $\mathit{node}(x \cdot c) = \mathit{node}(x \cdot i)$ and $\mathit{ri}(x \cdot c) = d_i$ hold by definition of r. By induction, $\mathit{ti}(\mathit{node}(x)) = d$, and thus $\mathit{ti}(\mathit{node}(x) \cdot i) = d_i$ by definition of t, which concludes the proof of the claim.

Now we can prove that r is a run on t, according to Definition 4.11: consider a node x with $r(x) = (v, q)$. Then there is an individual $d \in \Delta^{\mathcal{I}}$ with $d = \mathit{ri}(x)$ and $d \in \mathit{tr}(q)^{\mathcal{I}}$. Set $\mathsf{Q} = \mathit{tr}(q)$. Moreover, by definition, for $\mathit{letter}(t(v)) = \mathsf{A}$, we have $t(v) = \mathit{tr}^{-1}(\mathsf{A})$. The claim yields $t(v) = r(x) = d$, which implies $\mathsf{A} = \mathit{letter}(d)$ by construction. Summing up, we obtain $d \in \mathsf{A}^{\mathcal{I}} \cap \mathsf{Q}^{\mathcal{I}}$.

Since $\mathcal{I}$ is a model of $\mathit{tr}(\mathcal{A})$, $d \in \mathsf{C}^{\mathcal{I}}$ holds for $\mathsf{A} \sqcap \mathsf{Q} \sqsubseteq \mathsf{C}$, the translation of $\delta(\mathit{tr}^{-1}(\mathsf{A}), q) = \varphi$. As d is an instance of C, there exists a set $N = \{n_1, \ldots, n_\ell\} \subseteq L$ which "satisfies" C. For every $n_i, 1 \leq i \leq \ell$, we define a $p_i \in K_0 \times Q$ as follows:

- if $n_i = \exists \mathsf{f}_c.\mathsf{Q}$ for some f_c, Q, then d has an f_c-successor $d_c \in \mathsf{Q}^{\mathcal{I}}$. By construction of t, v has a c-successor $v \cdot c$, and x has a successor $x \cdot c'$ with $r(x') = (v \cdot c, \mathit{tr}^{-1}(\mathsf{Q}))$. We set $p_i := (c, \mathit{tr}^{-1}(\mathsf{Q}))$;
- if $n_i = \mathsf{Q}$ for some Q, then $d \in \mathsf{Q}^{\mathcal{I}}$. By construction, x has a successor $x \cdot j$ with $r(x \cdot j) = (v, q)$. We set $p_i := (0, q)$.

It can easily be seen that set $S := \{p_i \mid 1 \leq i \leq \ell\}$ satisfies $\varphi := \mathit{tr}^{-1}(\mathsf{C})$, and therefore Condition 1 from Definition 4.11 is satisfied. Condition 2 holds by construction of S, thus r is a run. Additionally, r is successful by the definition of $r(\varepsilon)$, which concludes the proof of the "if"-direction. □

Lemma 5.4 has two consequences: firstly, the emptiness of a language given by an alternating automaton (and thus reasoning problems for various logics) can be decided by translating it into an $\mathcal{ELU}_f$-concept and TBox and then deciding their satisfiability using one of the existing DL systems, e. g. FaCT or Racer. Secondly, we have obtained tight complexity bounds for $\mathcal{ELU}_f$: in Section 4.4, we have shown how alternating automata can be used to decide $\mathcal{ALC}$ concept satisfiability w. r. t. general TBoxes, which is an ExpTime-hard problem (shown by transfer from $\mathcal{ALU}$

(Calvanese, 1996)). Since the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is polynomial in the size of C and $\mathcal{T}$, this yields ExpTime-hardness of the emptiness problem for alternating one-way automata, and the translation into $\mathcal{ELU}_f$ being polynomial implies that satisfiability of $\mathcal{ELU}_f$-concepts w. r. t. TBoxes is ExpTime-hard, in contrast to $\mathcal{EL}$, for which satisfiability w. r. t. TBoxes is still polynomial (see Section 2.3.4).[2]

Finally, $\mathcal{ELU}_f$ is a fragment of deterministic propositional dynamic logic which is in ExpTime (Ben-Ari, Halpern, and Pnueli, 1982) and allows for the internalisation of TBoxes (see e. g. Calvanese et al., 1999). Thus we have tight complexity bounds.

Corollary 5.6. Satisfiability of $\mathcal{ELU}_f$-concepts w. r. t. general TBoxes is ExpTime-complete.

This concludes our treatment of one-way automata. We have seen that the translation of the transition function is rather intuitive and only requires a comparably inexpressive DL. The translation of two-way automata, which is presented in the following section, is based on the same ideas, but capturing transitions to the father node requires a more involved construction and also a more expressive DL.

5.2 Translation of Two-Way Automata into $\mathcal{FLEUI}_f$

In this section, we extend the translation in Definition 5.2 from one-way to two-way automata. For (-1)-transitions to the father node, we require the *inverses* of the features f_i (see Figure 2.1). However, several issues about inverses have to be taken into account: firstly, the inverse of a feature need not be functional, e. g. the concept $\exists \mathsf{f}_1^-.\mathsf{Q}_1 \sqcap \exists \mathsf{f}_1^-.\mathsf{Q}_2$ can have an instance d with two f_1 predecessors that are labelled with Q_1 and Q_2, respectively. Therefore, we cannot simply translate a (-1)-transition into an existential restriction involving the inverse of a feature. Secondly, for a particular node in the completion tree, we do not know the label of the edge by which it is connected with the father node; thus we do not know which feature f_i to use in the translation of $(-1, q_0)$ into $\exists \mathsf{f}_i^-.\mathsf{Q}_0$. Both of these problems can be solved by using *value restrictions* instead of existential restrictions and thus enforcing that *all* predecessors for all features satisfy the corresponding concept.

Since all nodes except the root node in a completion tree are created because of an existential restriction in the predecessor node and all existential restrictions appearing in our translation involve a feature (and not its inverse), all of these nodes have exactly one predecessor for exactly one feature, thus this approach works for all nodes but the root node. There, a value restriction involving the inverse of a feature would be trivially satisfied, in contrast with the intended behaviour, namely being unsatisfiable, since a transition to the father of the root node is impossible. We therefore have to find another way of preventing these value restrictions for the root node: we use additional concept names R for the root node and NR appearing only in non-root nodes and the

[2] Recently, Baader et al. (2005) have shown that $\mathcal{EL}$ augmented with either disjunction or functional roles becomes ExpTime-hard.

GCI $\mathsf{R} \sqcap \mathsf{NR} \sqsubseteq \bot$, which ensures that no node can be labelled with both concepts. Then, we can translate a transition $(-1, q_i)$ into the concept

$$\mathsf{NR} \sqcap \bigsqcap_{j \in K} \forall \mathsf{f}_j^{-}.\mathsf{Q}_i,$$

and we reduce the emptiness problem of an alternating two-way automaton $\mathcal{A}$ to the satisfiability of the concept $\mathsf{R} \sqcap \mathsf{Q}_0$ w. r. t. to the TBox resulting from the translation of the transition relation of $\mathcal{A}$.

The DL that is obtained from extending $\mathcal{ELU}_f$ with inverse roles and value restrictions is called $\mathcal{FLEUI}_f$. This logic is still a fragment of $\mathcal{SHIQ}$ and therefore can be decided using the implementations mentioned in the previous section.[3]

Definition 5.7 ($\mathcal{FLEUI}_f$, translation tr′ into $\mathcal{FLEUI}_f$). For a feature f, f^{-} is a called an *inverse feature*. The set of $\mathcal{FLEUI}_f$ *concepts* is defined like the set of $\mathcal{ELU}_f$ concepts with the following addition: if f is a feature or an inverse feature and C is a concept, $\forall \mathsf{f}.\mathsf{C}$ and $\exists \mathsf{f}.\mathsf{C}$ are also concepts. The semantics of the additional constructors is defined as in Figure 2.1.

The translation $tr'(\mathcal{A})$ of a two-way automaton $\mathcal{A}$ into $\mathcal{FLEUI}_f$ is defined like the translation tr in Definition 5.2 with the following additions:

- $tr'(\mathcal{A})$ additionally contains the GCI $\mathsf{R} \sqcap \mathsf{NR} \sqsubseteq \bot$, where R and NR are concept names that do not appear in the translation of states and alphabet symbols;
- (-1)-transitions are translated as follows: $tr'(-1, q) = \mathsf{NR} \sqcap \bigsqcap_{i \in K} \forall \mathsf{f}_i^{-}.tr'(q)$. $\diamond$

The following lemma uses this translation to reduce the emptiness problem of looping alternating two-way automata to $\mathcal{FLEUI}_f$ concept satisfiability w. r. t. general TBoxes.

Lemma 5.8. The language accepted by a two-way alternating automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ is non-empty iff the concept $\mathsf{R} \sqcap tr'(q_0)$ is satisfiable w. r. t. the TBox $tr'(\mathcal{A})$.

Proof. For the "only-if" direction, we show that the interpretation $\mathcal{I}$ defined as in the proof of Lemma 5.4 also satisfies the GCIs involving value restrictions. We define the interpretation of the new concept names as $\mathsf{R}^{\mathcal{I}} = \{\varepsilon\}$ and $\mathsf{NR}^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \{\varepsilon\}$. Let t be the tree and r be the run used to define $\mathcal{I}$. Observe that the features f_c induce a tree-shaped structure over $\Delta^{\mathcal{I}}$ since the interpretations are defined by using the father-child relation in t. Consequently, every individual $v \in \Delta^{\mathcal{I}} \setminus \{\varepsilon\}$ has exactly one predecessor for exactly one feature.

Now let v be an individual with $v \in \mathsf{Q}_i^{\mathcal{I}} \cap \mathsf{A}_j^{\mathcal{I}}$ with the corresponding GCI $\mathsf{Q}_i \sqcap \mathsf{A}_j \sqsubseteq \mathsf{C}$ and transition $\delta(q_i, \sigma_j) = \varphi$. Since r is a run, there is a set $\{(a(1), q_{c(1)}), \ldots, (a(m), q_{c(m)})\}$ of literals which satisfies φ and for which there exist

[3] It is also possible to use role hierarchies instead of value restrictions to represent transitions to the father node: in this case, a "predecessor feature" f_{-1} is defined as a super-role of the inverses of all features f_i. This gives rise to a translation of alternating two-way automata into the DL $\mathcal{ELUHI}_f$.

appropriate nodes in r. If $a(i) = (-1)$ holds for some $1 \leq i \leq m$ then the corresponding node in r is labelled with $(v \cdot (-1), q_{c(m)})$ and consequently the individual $v \cdot (-1)$, which is the unique predecessor of v, is labelled with $\mathsf{Q}_{c(m)}$. Therefore v satisfies $\prod_{j \in K} \forall \mathsf{f}_j^-.\mathsf{Q}_{c(m)}$. Moreover, since r is a run and thus contains no (-1)-transitions from the root node of t, $v \in \mathsf{NR}^{\mathcal{I}}$ holds.

For the "if" direction, let $\mathcal{I}$ be a model for $\mathsf{R} \sqcap \mathsf{Q}_0$ w. r. t. $\mathit{tr}'(\mathcal{A})$. The tree t is defined as in the proof of Lemma 5.4, with the only difference that we use an individual d_ε from $\mathsf{R} \sqcap \mathsf{Q}_0$ to define the label of the root node. In order to define the the label for a node x of the run r, we have to consider the possibility that the individual $d = \mathit{ri}(x)$ may have several predecessors. For the labels for the children nodes of x, we use the individual that the (unique) predecessor of $\mathit{node}(x)$ refers to; formally: $d_{-1} := \mathit{node}(x) \cdot (-1)$, if $\mathit{node}(x) \neq \varepsilon$, and $d_{-1} := \#$, otherwise. As in the case of one-way automata, we label one successor of x with $(\mathit{node}(x) \cdot (-1), q_j)$ for every $q_j \in \mathit{states}(d_{-1})$. Then, the claim $\mathit{ri}(x) = \mathit{ti}(\mathit{node}(x))$ follows as before.

In order to show that r is a run, we now consider a run node x with $\mathit{ri}(x) = d$ and a transition $(-1, q)$ from x that translates into $\mathsf{NR} \sqcap \prod_{i \in K} \forall \mathsf{f}_i^-.\mathsf{Q}$. Since $\mathcal{I}$ is a model, d_x is not equal to d_ε (otherwise d_x would be contained in $\mathsf{R}^{\mathcal{I}} \cap \mathsf{NR}^{\mathcal{I}}$). It follows by construction of t that there exists at least one predecessor of d_x for some feature f_c (otherwise d_x would not appear in the range of ti and ri), and that one of these predecessors relates to the predecessor of $\mathit{node}(x)$. Moreover, all predecessors belong to $\mathsf{Q}^{\mathcal{I}}$, and together with the claim above, this yields that one child node of x is labelled with $(\mathit{node}(x) \cdot (-1), q)$. Consequently, r is a successful run of $\mathcal{A}$ on t. □

Comparing a model $\mathcal{I}$ for $\mathit{tr}'(\mathcal{A})$ (as in Figure 5.1) with a strategy tree s for $\mathcal{A}$ (as in Definition 4.13), it turns out that they are "almost" identical. More precisely, the *unravelling* of $\mathcal{I}$ looks exactly like s with *all dummy nodes removed*: in both cases, we obtain a tree in which every node is labelled with exactly one alphabet symbol and a set of states. The alphabet symbols give rise to a tree that is accepted by $\mathcal{A}$, and the states in which $\mathcal{A}$ operates on a node v are contained in the label of v. Thus, the intuitive translation of an alternating automaton into a DL leads to the result that the tableau algorithm constructs a model that the automata algorithm accepts. This illustrates the close relationship between tableaus and alternating automata, and it supports the claim that alternating automata allow for a particularly "natural" way of handling DLs.

This sums up the theoretical results of our translation. In order to test its practical relevance, i. e. to to examine if the optimisations of tableau algorithms can compensate for the overhead introduced by the automata construction, we evaluated the performance of existing tableau-based DL reasoners on TBoxes resulting from the translation of alternating two-way automata into $\mathcal{FLEUI}_f$ in Definition 5.7. The automata that were the source for the translation are described in the following section.

5.2.1 An ATA Algorithm for $\mathcal{ALCIO}$

The automata we used for our experiments result from a decision procedure for the DL $\mathcal{ALCIO}$, i. e. $\mathcal{ALC}$ with inverse roles and nominals (see Figure 2.2). Sattler and

Vardi (2001) describe an algorithm using two-way alternating tree automata to decide satisfiability of formulas in the hybrid μ-calculus, a modal logic which corresponds to $\mathcal{ALCIO}$ extended with fixpoints and the universal role. Extending the translation of $\mathcal{ALC}$ to one-way automata in Definition 4.17 to inverse roles is comparably easy: it requires (-1)-transitions to properly handle value restriction and additional role symbols $\mathsf{s}_{\mathsf{r}^-}$ in RC, indicating that a node is an r^--successor.

Capturing nominals is more involved: since every node can be labelled with a nominal, but every nominal may only be satisfied by one individual, $\mathcal{ALCIO}$ does not have the tree model property. However, a model for an input $(\mathsf{C}, \mathcal{T})$ containing ℓ nominals $\{\mathsf{N}_1, \ldots, \mathsf{N}_\ell\}$ can be decomposed into a forest of $\ell + 1$ trees by "cutting off" subtrees whose roots are labelled with nominals. Thus the initial concept C and each N_i are represented by one of the trees. Uniqueness of the nominals is ensured by allowing nodes labelled with nominals only at the roots of these trees; at lower levels, a node v which requires a successor labelled with a nominal is labelled with an additional alphabet symbol $\xrightarrow{\mathsf{r}} \mathsf{N}_i$, indicating that it has N_i as an r-successor. A model can then be generated by introducing an r-edge between v and the root of the tree representing N_i. In order to ensure that the nominal nodes appearing at lower levels in the tree are labelled with the same concepts as the nominal nodes, this algorithm uses a *guess* $\mathcal{G}$ of the types of the nominals and, since we do not impose the unique name assumption for nominals, a mapping f between nominals to capture the possibility that one individual represents two nominals. In addition, it is necessary to guess the roles connecting two nominals to ensure that theses roles are represented consistently in the two types.

Definition 5.9 (Guess $\mathcal{G}$). Let $\mathsf{N}_\mathsf{R}, \mathsf{N}_\mathsf{C}$, and $\mathsf{N}_\mathsf{O} \subseteq \mathsf{N}_\mathsf{C}$ be sets of role names, concept names, and nominal names, respectively. For an $\mathcal{ALCIO}$ concept C and a TBox $\mathcal{T}$, let $\mathsf{N}_\mathsf{O}(\mathsf{C}, \mathcal{T}) = \{\mathsf{N}_1, \ldots, \mathsf{N}_\ell\}$ be the set of all nominals occurring in C or $\mathcal{T}$, and let $\mathsf{N}_\mathsf{R}(\mathsf{C}, \mathcal{T})$ be the set consisting of all occurring role names and their inverses. A *guess* $\mathcal{G} = (G, f, C)$ consists of

- a *guess list* $G = (\gamma_1, \ldots, \gamma_\ell)$,
- a set of *connections* $C \subseteq \mathsf{N}_\mathsf{O}(\mathsf{C}, \mathcal{T}) \times \mathsf{N}_\mathsf{R}(\mathsf{C}, \mathcal{T}) \times \mathsf{N}_\mathsf{O}(\mathsf{C}, \mathcal{T})$, and
- a *guess mapping* $f : \{1, \ldots, \ell\} \to \{1, \ldots, \ell\}$

such that the following conditions are satisfied:

1. for each $1 \leq i, j \leq \ell$, we have $\emptyset \subsetneq \gamma_i \subseteq \mathit{sub}(\mathsf{C}, \mathcal{T})$ or $\gamma_i = \#$,
2. $\mathsf{N}_i \in \gamma_{f(i)}$, $\mathsf{N}_i \notin \gamma_j$ for $f(i) \neq j$, $\mathsf{N}_\mathsf{O} \cap \gamma_i = \emptyset$ implies $\gamma_i = \#$, and
3. $(\mathsf{N}_i, \mathsf{r}, \mathsf{N}_j) \in C$ iff $(\mathsf{N}_j, \mathsf{r}^-, \mathsf{N}_i) \in C$. ◇

The intuition behind this definition is that γ_i contains all the concepts from $\mathit{sub}(\mathsf{C}, \mathcal{T})$ that the (unique) individual representing the i-th nominal satisfies (Condition 1); therefore it includes N_i. In order to allow for one individual satisfying several

nominals, we use the mapping f: for example, to express that the individual d belongs to the interpretations of both N_1 and N_3, one can define $f(3) = 1$, $\gamma_3 = \#$, and $\gamma_1 = \{\mathsf{N}_1, \mathsf{N}_3\}$ (Condition 2). Finally, the set of connections C and the corresponding restrictions on the γ sets ensures that d_i (the individual representing N_i) is an r-successor of d_j (the individual representing N_j) iff d_j is an r^--successor of d_i (Condition 3).

In order to test for the existence of a forest as described in the beginning of this section with a tree automaton, the forest is transformed back into a tree by adding a new root node (labelled with a new alphabet symbol ROOT) and $n+1$ children. Such a tree is called *pre-model* for $(\mathsf{C}, \mathcal{T}, \mathcal{G})$. Testing if a tree t is a pre-model for the input $(\mathsf{C}, \mathcal{T}, \mathcal{G})$ requires two automata: the first one, $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$, is a modification of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ in Definition 4.17, where the transition relation is extended to deal with inverse roles and nominals. The second one, $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$, tests if the structure of the tree is as defined above, i. e. if the tree can in fact be regarded as a pre-model consisting of several trees. If t is accepted by both $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$ and $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$, it is a pre-model for $(C, \mathcal{T}, \mathcal{G})$. An alternating automaton accepting the intersection of the languages accepted by $\mathcal{A}$ and $\mathcal{A}'$ is easy to construct: one simply introduces a new initial state from which one copy is sent to each of the initial states of $\mathcal{A}$ and $\mathcal{A}'$.

In the following, we describe these two automata in detail. In $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$, starting from the root node, we ensure that each of the nominal trees can serve as a model for the guess of the corresponding nominal and that one of the children is a model for C. Similar to the state GCI, the state NOM ensures that if a node v is reachable via a role r from a nominal node v_i whose label contains a value restriction $\forall\mathsf{r}.\mathsf{D}$, then v's label contains D. Existential and value restrictions that involve dealing with nominals require comparing the restriction with the guess $\mathcal{G}$: if v's label contains the concept $\exists\mathsf{r}.\mathsf{D}$, the nominal node v_i is an r-successor of v, and the corresponding γ set contains D, then the existential restriction is satisfied, and it is not necessary to ensure that one of v's r-successors in the tree is labelled with D. Similarly, if v's label contains $\forall\mathsf{r}.\mathsf{D}$ and v_i is an r-successor, then there is no transition from the current state if D is not contained in the guess for the corresponding nominal.

Definition 5.10 (Automaton $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$). For the sets N_C, N_R and N_O defined as in Definition 5.9 and an $\mathcal{ALCIO}$ concept C and TBox $\mathcal{T}$, the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}} = (Q, \Sigma, q_0, \delta)$ is defined as in Definition 4.17, with the following modifications:

- the arity k of the input tree is the maximum of $\#\mathsf{N}_\mathsf{O}(\mathsf{C}, \mathcal{T}) + 1$ and the number of existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$;
- Q additionally contains, for each $\mathsf{r} \in \mathsf{N}_\mathsf{R}(\mathsf{C}, \mathcal{T})$, the states $\mathsf{s}_{\mathsf{r}^-}$ and $\overline{\mathsf{s}}_{\mathsf{r}^-}$;
- Σ additionally contains ROOT, and each set $\sigma \in \Sigma$ can additionally contain, for every $\mathsf{r} \in \mathsf{N}_\mathsf{R}(\mathsf{C}, \mathcal{T})$ and $\mathsf{N} \in \mathsf{N}_\mathsf{O}(\mathsf{C}, \mathcal{T})$, the symbols $\xrightarrow{\mathsf{r}} \mathsf{N}$ and $\mathsf{s}_{\mathsf{r}^-}$;
- δ is modified as described in Figure 5.3. ◇

The second automaton $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$ ensures that the nominal nodes are labelled with all concepts in the corresponding guess (states Q_i), that the nodes in the first level do

$$
\begin{aligned}
\Gamma_i &= \begin{cases} (i,\#) & \text{if } \gamma_i = \# \\ \bigwedge_{\mathsf{D}\in\gamma_i}(i,\mathsf{D}) & \text{otherwise} \end{cases} \\
N_\sigma &= \bigwedge_{\xrightarrow{\bar{\mathsf{r}}}\mathsf{N}_i\in\sigma \text{ and } \forall \mathsf{r}.\mathsf{D}\in\gamma_{f(i)}} (0,\mathsf{D}) \\
\delta(\mathit{START},\sigma) &= \bigwedge_{i=1}^{\ell}\Gamma_i \wedge \bigvee_{i=1}^{k}(i,\mathsf{C}) \wedge \bigwedge_{i=1}^{k}((i,\#)\vee((i,\mathit{GCI})\wedge(i,\mathit{NOM}))) \\
\delta(\mathit{NOM},\sigma) &= N_\sigma \wedge \bigwedge_{i=1}^{k}((i,\mathit{NOM})\vee(i,\#)) \\
\delta(\exists \mathsf{r}.\mathsf{D},\sigma) &= \begin{cases} \mathit{true} & \text{if } \xrightarrow{\mathsf{r}}\mathsf{N}_i\in\sigma \text{ and } \mathsf{D}\in\gamma_{f(i)} \\ \bigvee_{i=1}^{k}((i,\mathsf{s}_\mathsf{r})\wedge(i,\mathsf{D})) & \text{otherwise} \end{cases} \\
\delta(\forall \mathsf{r}.\mathsf{D},\sigma) &= \begin{cases} \mathit{false} & \text{if } \xrightarrow{\mathsf{r}}\mathsf{N}_i\in\sigma \text{ and } \mathsf{D}\notin\gamma_{f(i)} \\ ((-1,\mathsf{D})\vee(0,\bar{\mathsf{s}}_{\bar{\mathsf{r}}}))\wedge & \text{otherwise} \\ \bigwedge_{i=1}^{k}((i,\bar{\mathsf{s}}_\mathsf{r})\vee(i,\mathsf{D})\vee(i,\#)) & \end{cases}
\end{aligned}
$$

Figure 5.3: $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$ transition relation

not contain the role symbols s_r or $\mathsf{s}_{\bar{\mathsf{r}}}$ (state *FIRST*), that *ROOT* only appears at the root level, and that nominals only appear at the first level (state *RN*).

Definition 5.11 (Automaton $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$). Let C, $\mathcal{T}$, $\mathcal{G}$ and ℓ be as in Definition 5.10. The automaton $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$ is defined as $(Q',\Sigma,\mathit{START}',\delta')$ with $Q' = \{\mathit{START}',\mathit{FIRST},\mathit{RN},Q_1,\ldots,Q_\ell\}$ and δ' as shown in Figure 5.4. ◇

Thus, we have an ATA decision procedure for satisfiability in $\mathcal{ALCIO}$ which, together with our translation of alternating automata into $\mathcal{FLEUI}_f$, enables us to use reasoners for the logic $\mathcal{SHIQ}$ to perform reasoning in a language allowing for nominals. The chain chain of translations is illustrated in the following:

$$
\mathcal{ALCIO} \xrightarrow[\substack{\text{linear in } \#Q \\ \text{exponential in } \#\Sigma}]{\text{Sattler and Vardi (2001)}} \text{ATA} \xrightarrow[\substack{\text{polynomial in} \\ \#Q+\#\Sigma}]{\text{Def. 5.7}} \mathcal{FLEUI}_f \rightsquigarrow \text{Reasoner}
$$

Although the size of the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T},\mathcal{G}}$, i. e. the number of its states, is polynomial in the size of the input $(\mathsf{C},\mathcal{T},\mathcal{G})$, the size of the automaton's transition function is exponential since there is one alphabet symbol for every possible set of concept names. Together with our linear translation in Section 5.2, this yields a TBox whose size is also exponential in the size of $(\mathsf{C},\mathcal{T},\mathcal{G})$.

5.2.2 Test Concepts

We have prototypically implemented the automata algorithm for $\mathcal{ALCIO}$ described in the Definitions 5.10 and 5.11 and the translation of alternating automata into $\mathcal{FLEUI}_f$

$$
\begin{array}{rcl}
\delta'(\mathit{START}', \sigma) & = & \begin{cases} \bigwedge_{i=1}^{\ell}(i, Q_i) \wedge \bigwedge_{i=\ell+1}^{k}(i, \mathit{RN}) \wedge \bigwedge_{i=1}^{k}(i, \mathit{FIRST}) & \text{if } \sigma = \mathit{ROOT} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\delta'(\mathit{FIRST}, \sigma) & = & \begin{cases} \mathit{true} & \text{if } \sigma \cap \{\mathsf{s_r}, \mathsf{s_{r^-}} \mid \mathsf{r} \in \mathsf{N_R}\} = \emptyset \\ \mathit{false} & \text{otherwise} \end{cases} \\
\text{for } 1 \leq i \leq \ell: & & \\
\delta'(Q_i, \sigma) & = & \begin{cases} \bigwedge_{i=1}^{k}(i, \mathit{RN}) & \text{if } \sigma \cap \mathsf{N_C} = \gamma_i \cap \mathsf{N_C}, \sigma \neq \mathit{ROOT}, \text{ and,} \\ & \text{for each } (\mathsf{N}, \mathsf{r}, \mathsf{N}') \in C \text{ with } \mathsf{N} \in \sigma, \\ & \sigma \text{ contains } \xrightarrow{\mathsf{r}} \mathsf{N}' \\ \mathit{false} & \text{otherwise} \end{cases} \\
\delta'(\mathit{RN}, \sigma) & = & \begin{cases} \bigwedge_{i=1}^{k}(i, \mathit{RN}) & \text{if } \sigma \cap \mathsf{N_O} = \emptyset \text{ and } \sigma \neq \mathit{ROOT} \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.4: $\mathcal{A}'_{\mathsf{C},\mathcal{T},\mathcal{G}}$ transition relation

from Definition 5.7 in Lisp. As input for the translation, we used the empty TBox and the $\mathcal{ALCIO}$ concepts shown in Figure 5.5. Here, the expression $(\forall \mathsf{r})^i$ stands for $\underbrace{\forall \mathsf{r} \ldots \forall \mathsf{r}}_{i \text{ times}}$. The tests were performed with the concepts for $i \in \{0, \ldots, 5\}$ for every pattern. For satisfiable concepts containing the nominal N, the guess $\mathcal{G}$ was defined such that it assigned to N the smallest possible set of concepts that is required to ensure satisfiability. For unsatisfiable concepts, it assigned the set consisting of all required concepts except for the one leading to a contradiction. For example, the set γ_1 for the nominal N in the pattern ex-nc consists of N and A for both the satisfiable and the unsatisfiable variant.

The idea behind the structure of these patterns is the following:

- since nominals lead to a multiplication of the number of states, there are concept patterns sharing the same structure, but one pattern uses only a concept name ($\langle name \rangle$-c), the second one uses only a nominal name ($\langle name \rangle$-n), and the third one uses both ($\langle name \rangle$-nc), so that the influence of the mere existence of nominals can be tested;
- there are concepts exploiting the special features of nominals, i. e. the fact that there can be only one individual in the interpretation of a nominal (root-nc);
- there are concepts exploiting the interaction between a value restriction and and an existential restriction (all-c/nc) as well as the interaction between a role and its inverse (all-inv-c/nc).

5.2.3 Empirical Results

We tested our concepts with FaCT version 2.31.7 and Racer version 1.6.7 under Linux on the following hardware: Pentium-IV 1.7GHz, 512MB RAM, 1.5GB swap-

Name	Satisfiable Concept	Unsatisfiable Concept
ex-c	$(\exists \mathsf{r})^i.\mathsf{A}$	$(\exists \mathsf{r})^i.(\mathsf{A} \sqcap \neg\mathsf{A})$
ex-n	$(\exists \mathsf{r})^i.\mathsf{N}$	$(\exists \mathsf{r})^i.(\mathsf{N} \sqcap \neg\mathsf{N})$
ex-nc	$(\exists \mathsf{r})^i.(\mathsf{A} \sqcap \mathsf{N})$	$(\exists \mathsf{r})^i.(\mathsf{A} \sqcap \neg\mathsf{A} \sqcap \mathsf{N})$
all-c	$(\exists \mathsf{r})^i.\mathsf{A} \sqcap (\forall \mathsf{r})^i.\mathsf{B}$	$(\exists \mathsf{r})^i.\mathsf{A} \sqcap (\forall \mathsf{r})^i.\neg\mathsf{A}$
all-nc	$(\exists \mathsf{r})^i.\mathsf{A} \sqcap (\forall \mathsf{r})^i.\mathsf{N}$	$(\exists \mathsf{r})^i.\mathsf{A} \sqcap (\forall \mathsf{r})^i.(\mathsf{N} \sqcap \neg\mathsf{A})$
all-inv-c	$\mathsf{B} \sqcap (\exists \mathsf{r})^i.(\forall \mathsf{r}^-)^i.\mathsf{A}$	$\neg\mathsf{A} \sqcap (\exists \mathsf{r})^i.(\forall \mathsf{r}^-)^i.\mathsf{A}$
all-inv-nc	$\mathsf{N} \sqcap (\exists \mathsf{r})^i.(\forall \mathsf{r}^-)^i.\mathsf{A}$	$(\mathsf{N} \sqcap \mathsf{A}) \sqcap (\exists \mathsf{r})^i.(\forall \mathsf{r}^-)^i.(\neg\mathsf{A})$
root-nc	$\mathsf{N} \sqcap \mathsf{A} \sqcap (\exists \mathsf{r})^i.(\mathsf{N} \sqcap \mathsf{B})$	$\mathsf{N} \sqcap \mathsf{A} \sqcap (\exists \mathsf{r})^i.(\mathsf{N} \sqcap \neg\mathsf{A})$

Figure 5.5: Test concepts

space. The Lisp system used is Allegro Common Lisp version 6.2 for FaCT and version 6.1 for Racer. Figure 5.6 shows for every concept pattern the maximum i for which the concept could be tested within the time limit of 1000 seconds. The adjacent column shows the reason why the test of the next harder concept failed: "T" stands for timeout, "M" for insufficient memory. The total in the bottom rows also includes the concepts for $i = 0$ and is therefore (by the number 8) higher than the sum of the above rows. Since tests with internalised vs. non-internalised TBox (see Section 2.3.6) indicated that internalisation leads to a significant speedup, all results presented in the following were produced with internalisation.

Comparing the $\langle name \rangle$-ex-c and $\langle name \rangle$-ex-n concepts, it is obvious that the overhead introduced by nominals is significant. The same holds for every $\langle name \rangle$-c concept in comparison with the corresponding $\langle name \rangle$-nc concept. Thus, even when the special properties of nominals are not exploited, the mere presence of nominals slows down the reasoners, and their optimisations are not able to compensate for the overhead introduced. The situation is even worse for the concept pattern root-nc, which really exploits the expressivity of nominals and therefore requires node labels to contain $\xrightarrow{\mathsf{r}} \mathsf{N}$ symbols, which interact with the guess. For this concept pattern, both reasoners could only process the trivial concept for $i = 0$.

Moreover, Figure 5.6 shows that unsatisfiable concepts are significantly harder to process than their satisfiable counterparts. This is characteristic for tableau algorithms (see e. g. Horrocks, 1997; Hladik, 2002) because, in order to recognise a satisfiable input, the algorithm only has to find one sequence of non-deterministic decisions (i. e. one path through the "outer" tree described in Section 3.2) that leads to a model, and the search for this path can be sped up by efficient heuristics. For unsatisfiable inputs however, all sequences of decisions have to be tested, i. e. the entire search tree has to be traversed. Comparing the performance of the two systems, one can see that FaCT can handle more concepts than Racer.

Next, we examine if the calculation time does indeed increase exponentially in the size of the input automaton or if this behaviour can be prevented by the optimisations of the tableau algorithms. To this end, Figure 5.7 displays the processing times in relation to the size of the input automaton's transition function on a logarithmic scale. Since it is impossible to recognise a particular behaviour (e. g. polynomial or exponential runtime) from two or fewer measuring points, the figure only includes

	FaCT				Racer			
Concept	sat		unsat		sat		unsat	
ex-c	5		2	T	4	T	1	T
ex-n	3	M	1	T	1	T	0	T
ex-nc	2	M	0	T	0	M	0	T
all-c	3	M	2	T	3	T	1	T
all-nc	0	M	0	T	0	M	0	M
all-inv-c	3	T	2	T	1	M	1	M
all-inv-nc	0	M	1	T	0	M	0	M
root-nc	0	M	0	T	0	M	0	M
Total	24		16		17		11	
	40				28			

Figure 5.6: Number of successful tests

formula patterns for which at least three concepts could be processed. Although there are only few measuring points per pattern, the nearly linear graphs suggest that the calculation time increases almost exponentially in the size of the transition function/TBox (which in turn is exponential in the size of the input $\mathcal{ALCIO}$ concept). This shows that the optimisations of tableau algorithms are not able to compensate for the overhead introduced by an automata algorithm.

Berardi, Calvanese, and de Giacomo (2001) observed a poor performance of FaCT and Racer on TBoxes resulting from an automatic translation of UML class diagrams and identified three possible reasons for this behaviour:

- terminological cycles including existential restrictions,
- inverse roles,
- functional restrictions combined with existential restrictions.

Clearly, our translation contains all three kinds of problematic properties: we extensively use features and their inverses, and terminological cycles involving features are also not rare because every automaton $\mathcal{A}_\mathsf{C}$ for a satisfiable $\mathcal{ALCIO}$ concept C contains cycles (otherwise, there would not exist an infinite run). Another possible reason is the fact that the concept names are *strongly connected* in the sense there is a GCI for every possible pair of an alphabet symbol and a state, and thus every concept name added to a node causes a new GCI to fire and therefore to add new existential restrictions to the node label.

The observation that several kinds automatic translations from other formalisms into DLs lead to TBoxes that are very hard to process for existing reasoners, which on the other hand often perform surprisingly well on knowledge bases from real-life applications (Horrocks, 1997), supports the hypothesis that one reason for the good performance of tableau algorithms on these TBoxes is that real-life knowledge bases only exploit a small fraction of the expressivity of the underlying languages.

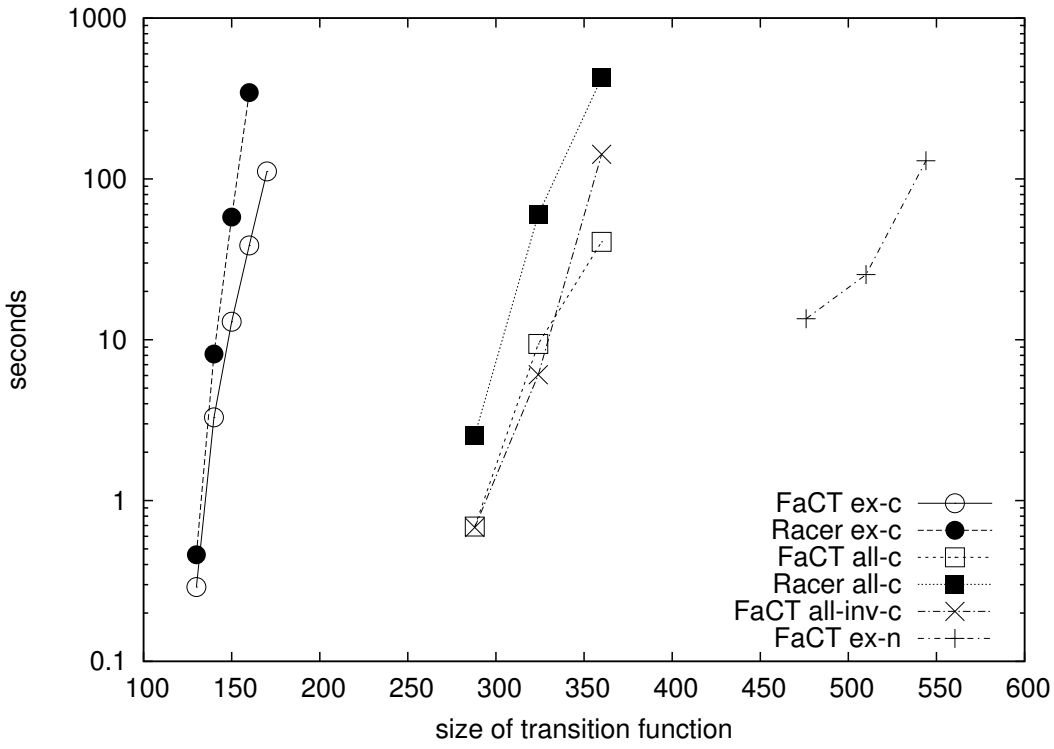

Figure 5.7: Runtimes of the satisfiability test for different patterns

5.3 Chapter Summary

In this chapter, we have developed methods for the translation of alternating automata into DL TBoxes. For one-way automata, the rather inexpressive logic $\mathcal{ELU}_f$ suffices: states and alphabet symbols are translated into concepts, and transitions to other states are encoded by existential restrictions involving features. For two-way automata, inverse roles and value restrictions are needed to capture transitions to the father node, thus target logic of the translation is $\mathcal{FLEUI}_f$.

In order to find out if the optimisations implemented in DL reasoners, which lead to good performance for many TBoxes resulting from real-life applications, are able to remedy the exponential blow-up induced by the automata algorithm, we have tested the behaviour of two DL reasoners on alternating automata which in turn result from a decision procedure for the logic $\mathcal{ALCIO}$.

From this effort, we have obtained three main results:

- The tableaus generated during the satisfiability test of these TBoxes closely resemble strategy trees for the corresponding automata, which again demonstrates the close relationship between two-way automata and tableau algorithms.
- Since our translation into $\mathcal{ELU}_f$ is linear and the emptiness problem of alternating one-way automata is ExpTime-hard, we obtain that concept satisfiability w. r. t. general TBoxes in the inexpressive DL $\mathcal{ELU}_f$ is ExpTime-complete.

- Empirical evaluation shows that the optimisations of the DL reasoners we used are not capable of compensating for the exponential blow-up introduced by the automata algorithm. The steep increase in processing time in relation to the size of the TBox supports previous observations that TBoxes resulting from automatic translation procedures are significantly harder to process than TBoxes resulting from real-life applications.

Chapter 6

PSPACE Automata

In Chapter 5 we have seen that the efficiency of tableau algorithms cannot easily be transferred to automata by simply translating an automaton into a DL for which satisfiability can be tested by a tableau algorithm, thus making use of the TA's efficient optimisations. In this chapter, we use a different approach to improve the efficiency of automata algorithms by employing methods stemming from the tableau paradigm: instead of first constructing the automaton and then testing emptiness with a TA, we modify the automata construction and the emptiness test itself using methods known from TAs. Our focus is also not on performance in practice, as in Chapter 5, but on the complexity class that can be obtained from an AA. As mentioned in Section 3.2.1, the TA for $\mathcal{ALC}$ concept satisfiability is in PSPACE because

- the maximum length of a path within the tableau is linear in the size of the input and
- it is only necessary to keep one path in memory at a time.

The reason for the polynomial length of a path in a tableau is the fact that the *role depth*, i. e. the maximum nesting depth of quantifiers, decreases from a node to its child. Since the role depth of concepts for the root node is obviously bounded by the size of the input, so is the maximum length of a path. From a model-theoretic point of view, one can say that the reason why $\mathcal{ALC}$ satisfiability is in PSPACE is that $\mathcal{ALC}$ has the *finite tree model property*, i. e. every satisfiable concept has a model which is a finite tree, and additionally that there is a polynomial bound for the maximum length of a path in such a tree.

In the first part of this chapter, we use this observation to define *segmentable automata*, a class of automata for which the state space is divided into a hierarchy of segments, where each transition leads to a lower segment within the hierarchy. If the number of segments is polynomial in the size of the input, we can show a PSPACE result with an argument that is similar to the one mentioned above for tableau algorithms. Using this approach, we can provide an alternative proof based on an automata algorithm for the known result that $\mathcal{ALC}$ concept satisfiability w. r. t. acyclic TBoxes is in PSPACE.

There are, however, DLs for which the satisfiability problem is decidable in PSPACE, but which do not have the *finite* tree model property. For example, in the DL $\mathcal{SI}$ (see Section 2.3.5), some concepts require cyclic or infinite models, but it is possible to define a TA for $\mathcal{SI}$ concept satisfiability that is still in PSPACE (Horrocks et al., 1999). This result is achieved by using the *blocking* technique (see Section 3.2.2): a new node is only generated if there exists no predecessor node with the same label. In the second part of this chapter, we adapt this technique by defining *blocking automata* for which there is a blocking relation between the states of the automaton. This allows us to obtain a new PSPACE result for satisfiability of $\mathcal{SI}$ concepts w. r. t. acyclic TBoxes.

For both approaches, the state space of the automaton under consideration is still exponential in the size of the input. It is therefore impossible to construct the entire automaton first and perform the bottom-up emptiness test afterwards as sketched in Section 4.1. Instead, we have to perform the emptiness test *top-down* and *interleave* it with the automaton's construction, i. e. construct the automaton *on-the-fly*. This is essential for obtaining a PSPACE result because it allows us to avoid generating states that are irrelevant for the result of the emptiness test. However, it also has two disadvantages: firstly, as the top-down emptiness test for automata is non-deterministic, we obtain a non-deterministic complexity class, NPSPACE. Since NPSPACE is equal to PSPACE by the theorem of Savitch (1970), this is not an essential problem for our approach. Secondly, the top-down emptiness test does not automatically terminate (see Section 4.2). We therefore have to halt the test after reaching the lowest level in the hierarchy (for segmentable automata) or a blocked state (for blocking automata).

This chapter is based on work that was previously published by Hladik and Peñaloza (2006); and Baader, Hladik, and Peñaloza (2006, 2007a, 2008).

6.1 Segmentable Automata

In this first framework, the conditions that an AA has to fulfil in order to provide a PSPACE upper bound are rather strong. Consequently, it is not too difficult to prove the complexity results for this framework and to apply it for a specific DL, but it also covers only a limited class of automata (and thus logics). In Section 6.2, we will define a more general framework that, as argued in Section 6.2.4, can be regarded as a generalisation of this one.

6.1.1 An Automata Algorithm for $\mathcal{ALC}$ Concept Satisfiability w. r. t. Acyclic and General TBoxes

For our aims in this chapter, we have to modify the definition of the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ in Definition 4.6 in two ways. Firstly, as indicated in Section 4.3.1, we will simplify it by dropping the alphabet Σ: there is an obvious redundancy in the definition of the transition relation because the first component (the initial state) is always identical to the second component (the alphabet symbol). One can therefore omit Σ and work on the (unique) infinite k-ary unlabelled tree as input, i. e. automata for satisfiable

inputs will accept exactly one input. This also means that the input tree does not correspond to a model anymore, but instead the *successful run* does, since it is a relabelling of the input tree where every node is labelled with the corresponding state of the automaton, which in turn is a Hintikka set. A looping automaton operating on the unlabelled k-ary tree then consists only of three parameters: $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, I, \Delta)$ with $\Delta \subseteq Q^{k+1}$.

Secondly, the automata algorithm from Definition 4.6 only deals with general TBoxes. Since satisfiability of $\mathcal{ALC}$ concepts w. r. t. general TBoxes is ExpTime-hard, we will not be able to show PSpace results with this automata definition, but we have to extend it in order to take advantage of the better computational properties of acyclic TBoxes. In fact, the definition of the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is only modified indirectly by a change in the definition of $\mathcal{T}$-expanded Hintikka sets. For an acyclic set of concept definitions of the kind $\mathsf{A} \doteq \mathsf{C}$ (see Definition 2.2), the following definition of expansion is sufficient:

Definition 6.1 ($\mathcal{T}$-expanded for acyclic and general TBoxes $\mathcal{T}$). For a TBox $\mathcal{T}$, a Hintikka set S is called *$\mathcal{T}$-expanded* if, for every GCI $\mathsf{C} \sqsubseteq \mathsf{D} \in \mathcal{T}$, it holds that $\dot{\neg}\mathsf{C} \sqcup \mathsf{D} \in S$ and, for every concept definition $\mathsf{A} \doteq \mathsf{C} \in \mathcal{T}$, it holds that $\mathsf{A} \in S$ implies $\mathsf{C} \in S$ and that $\neg\mathsf{A} \in S$ implies $\dot{\neg}\mathsf{C} \in S$.

The definitions of *$\mathsf{C}, \mathcal{T}$-compatible*, *Hintikka set*, and *Hintikka tree for C and $\mathcal{T}$* are extended accordingly. ◇

This technique of handling acyclic concept definitions is referred to as *lazy unfolding* since, in contrast to GCIs, acyclic concept definitions are used only if the defined concept name (or its negation) is explicitly present in a Hintikka set. Proceeding this way is possible because in the construction of a model from a successful run, we can determine for all individuals that do not contain A or $\neg\mathsf{A}$ for a defined concept name $\mathsf{A} \doteq \mathsf{C}$ if they belong to $\mathsf{A}^{\mathcal{I}}$ or $(\neg\mathsf{A})^{\mathcal{I}}$ by testing if they belong to $\mathsf{C}^{\mathcal{I}}$ or not. Due to the acyclicity of $\mathcal{T}$, this cannot lead to ambiguities or contradictions. Note that the size of Hintikka sets is still polynomial in the size of the input $(\mathsf{C}, \mathcal{T})$: it may additionally contain one possibly complex concept for every concept definition, whose size is trivially bounded by the size of the input, and thus also the number of additional subconcepts required by the conditions for Hintikka sets is linear in the size of the input.

In order to construct a model from a run, we therefore have to ensure that we know the definition of $\mathsf{C}^{\mathcal{I}}$ before defining the set $\mathsf{A}^{\mathcal{I}}$. This is made possible by the following definition, which takes concept definitions into account:

Definition 6.2 (Expanded role depth). For an $\mathcal{ALC}$ concept C and an acyclic TBox $\mathcal{T}$, the *expanded role depth* $\mathsf{rd}_{\mathcal{T}}(\mathsf{C})$ is inductively defined as follows:

- $\mathsf{rd}_{\mathcal{T}}(\mathsf{A}) = 0$ for primitive concept names A;
- $\mathsf{rd}_{\mathcal{T}}(\mathsf{A}) = \mathsf{rd}_{\mathcal{T}}(\mathsf{C})$ for concept definitions $\mathsf{A} \doteq \mathsf{C}$;
- $\mathsf{rd}_{\mathcal{T}}(\neg\mathsf{A}) = \mathsf{rd}_{\mathcal{T}}(\mathsf{A})$;
- $\mathsf{rd}_{\mathcal{T}}(\mathsf{D} \sqcap \mathsf{E}) = \mathsf{rd}_{\mathcal{T}}(\mathsf{D} \sqcup \mathsf{E}) = \max\{\mathsf{rd}_{\mathcal{T}}(\mathsf{D}), \mathsf{rd}_{\mathcal{T}}(\mathsf{E})\}$;

- $\mathsf{rd}_{\mathcal{T}}(\forall \mathsf{r}.\mathsf{D}) = \mathsf{rd}_{\mathcal{T}}(\exists \mathsf{r}.\mathsf{D}) = \mathsf{rd}_{\mathcal{T}}(\mathsf{D}) + 1$.

Please note that $\mathsf{rd}_{\mathcal{T}}(\mathsf{C})$ is polynomially bounded by the size of $\mathsf{C}, \mathcal{T}$. For a set of concepts S, $\mathsf{rd}_{\mathcal{T}}(S)$ is defined as $\max\{\mathsf{rd}_{\mathcal{T}}(\mathsf{D}) \mid \mathsf{D} \in S\}$. The set of *subconcepts of depth up to* n, $\mathsf{sub}_{\leqslant n}(\mathsf{C}, \mathcal{T})$, is defined as $\{\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid \mathsf{rd}_{\mathcal{T}}(\mathsf{D}) \leq \max\{0, n-1\}\}$.[1] ◇

Note that $\mathsf{rd}_{\mathcal{T}}$ is well-defined because $\mathcal{T}$ is acyclic. With the above modifications of the definitions of expanded Hintikka sets and Hintikka trees, we obtain a result analogous to Theorem 4.5. Since we prove a stronger result for an extension of $\mathcal{ALC}$ in Section 6.2.1, we omit the proof of the following theorem here.

Theorem 6.3. An $\mathcal{ALC}$ concept C is satisfiable w. r. t. a TBox $\mathcal{T}$ iff there is a $\mathsf{C}, \mathcal{T}$-compatible Hintikka tree t with $\mathsf{C} \in t(\varepsilon)$.

The automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is then defined as in Definition 4.6, omitting the alphabet Σ and using the modified definition of $\mathcal{T}$-expanded from definition 6.1. Soundness and completeness of the automata algorithm again follow directly from Theorem 6.3 and the fact that the transition relation Δ of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ uses the same notion of $\mathsf{C}, \mathcal{T}$-compatible as the Hintikka tree.

Lemma 6.4. The language accepted by the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is empty iff C is unsatisfiable w. r. t. $\mathcal{T}$.

Theorem 6.5. Satisfiability of $\mathcal{ALC}$ concepts w. r. t. general TBoxes is decidable in EXPTIME.

In the following, we show how this result can be improved for the special case of acyclic TBoxes.

6.1.2 The Framework for Segmentable Automata

In this section we develop a framework for automata in which the state space can be separated into a hierarchy of segments where every transition leads to a lower level in the hierarchy. If there are m segments and each element of the lowest segment can be the root of a successful run, then it is possible to restrict the emptiness test to runs of depth m. Formalising these conditions is the purpose of the following definition. In Section 6.1.3 we will then show how the role depth of concepts can be used to define such a hierarchy for the $\mathcal{ALC}$ automata algorithm.

Definition 6.6 (Q_0-looping, m-segmentable). Let $\mathcal{A} = (Q, \Delta, I)$ be a looping automaton on k-ary trees and $Q_0 \subseteq Q$. We call $\mathcal{A}$ Q_0-*looping* if for every $q \in Q_0$ there exists a set of states $\{q_1, \ldots, q_k\} \subseteq Q_0$ such that $(q, q_1, \ldots, q_k) \in \Delta$.

An automaton $\mathcal{A} = (Q, \Delta, I)$ is called m-*segmentable* if there exists a partition $Q_0, Q_1, \ldots, Q_m$ of Q such that $\mathcal{A}$ is Q_0-looping and, for every $(q, q_1, \ldots, q_k) \in \Delta$, it holds that if $q \in Q_n$, then $q_i \in Q_{<n}$ for $1 \leq i \leq k$, where $Q_{<n}$ denotes $Q_0 \cup \bigcup_{j=1}^{n-1} Q_j$.◇

[1] This seemingly over-complicated definition allows us to also use negative values for n and thus to avoid case distinctions in the following proofs.

```
 1: if I ≠ ∅ then
 2:    guess an initial state q ∈ I
 3: else
 4:    return "empty"
 5: end if
 6: if there is a transition from q then
 7:    guess such a transition (q, q_1, ..., q_k) ∈ Δ
 8:    push(SQ, (q_1, ..., q_k)), push(SN, 0)
 9: else
10:    return "empty"
11: end if
12: while SN is not empty do
13:    (q_1, ..., q_k) := pop(SQ), n := pop(SN) + 1
14:    if n ≤ k then
15:       push(SQ, (q_1, ..., q_k)), push(SN, n)
16:       if length(SN) < m − 1 then
17:          if there is a transition from q_n then
18:             guess a transition (q_n, q'_1, ..., q'_k) ∈ Δ
19:             push(SQ, (q'_1, ..., q'_k)), push(SN, 0)
20:          else
21:             return "empty"
22:          end if
23:       end if
24:    end if
25: end while
26: return "not empty"
```

Figure 6.1: Emptiness test for segmentable automata

Note that it follows immediately from this definition that for every element q of Q_0 there exists an infinite tree with q as root which is accepted by $\mathcal{A}$. The hierarchy $Q_m, \ldots, Q_0$ ensures that Q_0 is reached eventually.

We will now show that the emptiness test for an m-segmentable automaton can be performed using space logarithmic in the size of the automaton. In the following, when we speak about a *path of length* m in a k-ary tree, we mean a sequence of nodes $(v_1, \ldots, v_m) \in (K^*)^m$ such that v_1 is the root ε and v_{i+1} is a child of v_i.

The algorithm performing the emptiness test for m-segmentable automata is shown in Figure 6.1. Essentially, it performs a depth-first traversal of a successful run. Since $\mathcal{A}$ is m-segmentable, it is not necessary to descend to a depth larger than m. Moreover, since the different paths within the run are independent in the sense that the existence of transitions for one path does not affect the existence of transitions for other paths, the algorithm only has to keep one path in memory at a time. Note that the construction of the automaton is interleaved with the emptiness test, thus the

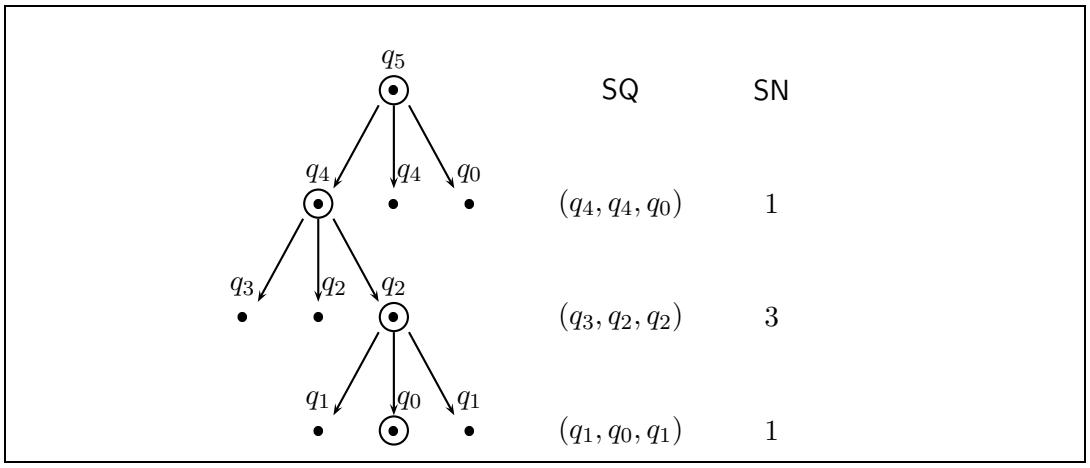

Figure 6.2: A successful run and the corresponding data structures

algorithm also never keeps the whole automaton in memory, but only the transitions that are relevant for the current path.

In order to remember the backtracking information for the depth-first traversal, we use two stacks: the stack SQ stores, for every node in the current path, the right-hand side of the transition which led to this node, and the stack SN stores, also for every node in the current path, on which component of this right-hand side the algorithm is currently working. If we refer to the depth of SN by d and to the elements in SN by $\mathsf{SN}(1)$[the bottom element], ..., $\mathsf{SN}(d)$[the top element], the next node to be checked is $\mathsf{SN}(1) \cdot \mathsf{SN}(2) \cdot \ldots \cdot \mathsf{SN}(d) + 1$. The entries of SQ and SN are elements of Q^k and K_0, respectively, and the number of entries is bounded by m for each stack.

Example 6.7 (On-the-fly emptiness test). Figure 6.2 shows the values stored in the stacks SQ and SN at the beginning of an iteration, and their relation with the traversal of the run. The circled nodes represent the path that was followed in order to reach the node about to be checked. The values of the elements of the stack are shown next to the depth in the run to which they correspond. For this reason, the stacks appear backwards, with their bottom element at the top of the figure, and vice versa.

After starting, the algorithm first guesses an initial state and a transition from that state. If it can find one, it pushes the labels of the nodes $1, \ldots, k$ onto stack SQ and the number 0 onto stack SN. Then it enters the while loop. As long as the stacks are not empty, the algorithm takes the top elements of both stacks. If $n > k$ in line 14, this indicates that it has checked all nodes on this level, and it backtracks without pushing anything on the stacks, which means that it will continue at the next upper level in the next loop. Otherwise, the algorithm stores the information that it has to check the next sibling by pushing the same tuple of states onto SQ and the incremented number n onto SN. Next, the algorithm tests if it has already reached the maximum depth in line 16. If the answer is yes, it backtracks, otherwise it tries to find a transition from the current node. If there is one, it pushes the required node

labels for the children of the current node onto SQ and the value 0 onto SN (line 18), which means that it will descend to the first child of the current node in the next loop. If there is no transition from the current state, the input is rejected.

Theorem 6.8. The emptiness problem of the language accepted by an m-segmentable automaton $\mathcal{A} = (Q, \Delta, I)$ over k-ary trees can be decided by a non-deterministic algorithm using space $O(\log(\#Q) \cdot m \cdot k)$, under the condition that it is possible to guess an initial state and a transition from a given state using space logarithmic in $\#Q$.

Proof. In order to show soundness, we will prove the claim "if the algorithm processes a node n (or backtracks without visiting n), then there is a successful run r in which n is labelled with the same state as in the algorithm (or a state $q \in Q_0$)" by induction over the iterations of the while loop. Initially, if the algorithm does not answer "empty", there is a transition $(q_0, q_1, \ldots, q_k)$ from an initial state, which can serve as root of the run r, and for which the states $1, \ldots, k$ of r can be labelled with $q_1, \ldots, q_k$.

If the algorithm has reached a node $n = n_0 \cdot n_1 \cdot \ldots \cdot n_\ell$ without answering "empty", it follows by induction hypothesis that each of the previously visited nodes corresponds to a node in r. Now there are two possibilities: firstly, if depth m has been reached then $r(n) \in Q_0$ holds because $\mathcal{A}$ is m-segmentable. Moreover, since $\mathcal{A}$ is Q_0-looping, there exists a k-ary run rooted at n in which all nodes are labelled with states from Q_0. Otherwise there is a transition $(r(n), q'_1, \ldots, q'_k)$ because the algorithm does not answer "empty", and we can use the same transition in the labelling of r.

In order to show completeness, we will prove the claim "if there exists a successful run, the algorithm can reach or skip every node in $\{1, \ldots, k\}^*$ without returning 'empty' as result" by induction over the structure of the run r. Since there is a successful run, we can guess an initial transition, and the nodes of the first level have the same labels in the algorithm as in r. If we have reached a node n that corresponds to the node n in r with $r(n) = q$, there are again two possibilities: if depth m has been reached, the algorithm will backtrack and skip over all successor nodes of n. Otherwise, since r is a successful run, there exists a transition $(q, q'_1, \ldots, q'_k)$, which the algorithm can guess, and therefore it will not return "empty".

Regarding memory consumption, observe that the stack SQ contains, for every level, k states, each of which can be represented using space logarithmic in the number of states, e. g. by using binary coding. Since we only descend into the depth m, there can be at most m tuples on the stack, thus the size of SQ is bounded by $\log(\#Q) \cdot m \cdot k$. SN stores at most m numbers between 0 and k. Since the guessing steps in the lines 2, 7, and 18 involve guessing one or $k + 1$ states, each of which can be represented using space logarithmic in the size of Q, the entire algorithm uses space logarithmic in the size of $\mathcal{A}$. □

The condition that an automaton $\mathcal{A}$ is m-segmentable is rather strong since it requires that from every state within the automaton, *only* transitions to states having a lower level in the hierarchy are possible. Therefore, the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ from Section 6.1.1 cannot easily be shown to be segmentable even if the TBox is empty because

$\mathcal{A}_{\mathsf{C},\mathcal{T}}$ does not *enforce* that the Hintikka sets of the successor states use only a lower quantification depth: for example, if the current state is $q_0 = \{\exists \mathsf{r}.\exists \mathsf{s}.\mathsf{A}\}$, then one possible successor state is $q_1 = \{\exists \mathsf{s}.\mathsf{A}\}$, but a transition to the state $q_2 = \{\exists \mathsf{s}.\mathsf{A}, \forall \mathsf{r}.\exists \mathsf{s}.\mathsf{B}\}$ is also possible if $\forall \mathsf{r}.\exists \mathsf{s}.\mathsf{B}$ is a subconcept of the input, and this transition does not decrease the maximum role depth. However, in order to test emptiness, we only need the *existence* of a transition leading to a lower class; therefore we will remove the transition involving q_2 from the transition relation and keep only the one to q_1. This means that the automaton does not accept all models for the input anymore, but it does not change the result of the emptiness test because the transition to q_1 is still present. This is the idea behind the generalisation in the following definition.

Definition 6.9 (Weakly-*m*-segmentable, reduced). An automaton $\mathcal{A} = (Q, \Delta, I)$ is called *weakly-m-segmentable* if there exists a partition $Q_0, Q_1, \ldots, Q_m$ of Q such that $\mathcal{A}$ is Q_0-looping and for every $q \in Q$ there exists a function $f_q : Q \to Q$ which satisfies the following conditions:

1. if $(q, q_1, \ldots, q_k) \in \Delta$, then $(q, f_q(q_1), \ldots, f_q(q_k)) \in \Delta$, and if $q \in Q_n$, then $f_q(q_i) \in Q_{<n}$ for all $1 \leq i \leq k$;
2. if $(q', q_1, \ldots, q_k) \in \Delta$, then $(f_q(q'), f_q(q_1), \ldots, f_q(q_k)) \in \Delta$.

If $\mathcal{A} = (Q, \Delta, I)$ is a weakly-m-segmentable automaton, then $\mathcal{A}^{\mathrm{r}}$, the *reduced automaton of* $\mathcal{A}$, is defined as follows:

$$\begin{array}{rcl} \mathcal{A}^{\mathrm{r}} & = & (Q, \Delta', I) \text{ with} \\ \Delta' & = & \{(q, q_1, \ldots, q_m) \in \Delta \mid \text{ if } q \in Q_n \text{ then } q_i \in Q_{<n} \text{ for } 1 \leq i \leq k\}. \end{array} \qquad \diamond$$

Note that the reduced automaton $\mathcal{A}^{\mathrm{r}}$ is m-segmentable by definition. In order to transform a tree accepted by $\mathcal{A}$ into one accepted by $\mathcal{A}^{\mathrm{r}}$, we replace, for all direct and indirect successors of a node labelled with q, the node labels q_i with $f_q(q_i)$. The conditions of Definition 6.9 guarantee that the tree obtained this way is accepted by $\mathcal{A}^{\mathrm{r}}$: intuitively, Condition 1 ensures that the class decreases for the transition from q to its direct successors, and Condition 2 ensures that there are still transitions for *all* successor nodes after modifying the node labels according to f_q. In order to show that we can restrict the emptiness test to $\mathcal{A}^{\mathrm{r}}$, the first thing to show is that the removed transitions are not required for the emptiness test, which is stated in the following lemma.

Lemma 6.10. Let $\mathcal{A} = (Q, \Delta, I)$ be a weakly-m-segmentable automaton. Then $\mathcal{L}(\mathcal{A})$ is empty iff $\mathcal{L}(\mathcal{A}^{\mathrm{r}})$ is empty.

Proof. Since every successful run of $\mathcal{A}^{\mathrm{r}}$ is also a successful run of $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ can only be empty if $\mathcal{L}(\mathcal{A}^{\mathrm{r}})$ is empty, thus the "only if" direction is obvious. For the "if" direction, we will show how to transform a successful run r of $\mathcal{A}$ into a successful run s of $\mathcal{A}^{\mathrm{r}}$. To do this, we traverse r breadth-first, creating an intermediate run $\hat{r}$, which initially is equal to r. At every node $v \in K^*$, we replace the labels of the direct and indirect successors of v with their respective $f_{\hat{r}(v)}$ values (see Definition 6.9). More formally,

at node v, we replace $\hat{r}(w)$ with $f_{\hat{r}(v)}(\hat{r}(w))$ for all $w \in \{v \cdot u \mid u \in K^+\}$ (recall that $K^+ = K^* \setminus \{\varepsilon\}$ by Definition 4.1).

By Definition 6.9, $\hat{r}$ is still a successful run after the replacement (here, Condition 2 is necessary to ensure transitions from the successors of v), and all direct successors of v are in a lower class than v (or Q_0 if $\hat{r}(v) \in Q_0$). Note that the labels of v's successors are not modified anymore after v has been processed. We can therefore define $s(n)$ as the value of $\hat{r}(v)$ "in the limit", i. e. after v has been processed. As argued before, s is a successful run in which every node is in a lower class than its father node (or both are in class 0). Consequently, all transitions used in s belong to the transition relation of $\mathcal{A}^{\mathrm{r}}$. □

Since the transitions from $\mathcal{A}$ that are not present in $\mathcal{A}^{\mathrm{r}}$ do not have an influence on the answer of the emptiness test, we can transfer the complexity result from segmentable to weakly-segmentable automata:

Corollary 6.11. The emptiness problem for a weakly-m-segmentable automaton $\mathcal{A} = (Q, \Delta, I)$ on k-ary trees can be tested using space $O(\log(\#Q) \cdot m \cdot k)$.

6.1.3 An Application to $\mathcal{ALC}$ with Acyclic TBoxes

In order to apply our framework to $\mathcal{ALC}$ with acyclic TBoxes, we will us the role depth to define the different classes, with the intuition that for a node q in a Hintikka tree, any concept having a higher role depth than q is superfluous in successors of q and thus the tuple without these concepts is also in the transition relation.

Lemma 6.12. Let C be an $\mathcal{ALC}$ concept, $\mathcal{T}$ an acyclic TBox, and $(S, S_1, \ldots, S_k)$ a C,$\mathcal{T}$-compatible tuple. Then, for $n = \mathsf{rd}_{\mathcal{T}}(S)$ and every $m \geq 0$, the following tuples are C,$\mathcal{T}$-compatible:

$$(S, \mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}) \cap S_1, \ldots, \mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}) \cap S_k)$$
$$(\mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S, \mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S_1, \ldots, \mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S_k)$$

Proof. We need to show that the conditions in Definitions 4.4 and 6.1 are satisfied for both tuples. In the case of the first tuple, suppose that $\exists \mathsf{r}.\mathsf{D} \in S$. Then, $S_{\varphi(\exists \mathsf{r}.\mathsf{D})}$ contains D and every concept E_i for which there is a value restriction $\forall \mathsf{r}.\mathsf{E}_i \in S$. But since $\mathsf{rd}_{\mathcal{T}}(\mathsf{D}) < \mathsf{rd}_{\mathcal{T}}(\exists \mathsf{r}.\mathsf{D}) \leq n$ and $\mathsf{rd}_{\mathcal{T}}(\mathsf{E}_i) < \mathsf{rd}_{\mathcal{T}}(\forall \mathsf{r}.\mathsf{E}_i) \leq n$, it holds that $\mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}) \cap S_{\varphi(\exists \mathsf{r}.\mathsf{D})}$ contains D and each of the E_i concepts.

For the second tuple, if we have $\exists \mathsf{r}.\mathsf{D} \in \mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S$, then it holds that $\mathsf{rd}_{\mathcal{T}}(\exists \mathsf{r}.\mathsf{D}) < m$ and $\mathsf{D} \in S_{\varphi(\exists \mathsf{r}.\mathsf{D})}$. If additionally there is a concept term E_i such that the value restriction $\forall \mathsf{r}.\mathsf{E}_i \in \mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S$, then again $\mathsf{rd}_{\mathcal{T}}(\mathsf{E}_i) < m$ and $\mathsf{E}_i \in S_{\varphi(\exists \mathsf{r}.\mathsf{D})}$. Hence, $\mathsf{sub}_{\leqslant m-1}(\mathsf{C}, \mathcal{T}) \cap S_{\varphi(\exists \mathsf{r}.\mathsf{D})}$ contains D and each such concept E_i.□

Using Lemma 6.12, we can apply the framework of segmentable automata to the automata algorithm for $\mathcal{ALC}$ concept satisfiability w. r. t. acyclic TBoxes:

Theorem 6.13. Let C be an $\mathcal{ALC}$ concept, $\mathcal{T}$ an acyclic TBox and $m = \mathsf{rd}_{\mathcal{T}}(\mathsf{C})$. Then $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, \Delta, I)$ is weakly-m-segmentable.

Proof. We have to give the segmentation of Q and the functions f_q and show that they satisfy the conditions in Definition 6.9. The classes Q_i for $0 \leq i \leq m$ and the functions f_q for $q, q' \in Q$ are defined as follows:

$$\begin{aligned} Q_i &:= \{S \in Q \mid \mathsf{rd}_{\mathcal{T}}(S) = i\} \\ f_q(q') &:= q' \cap \mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}), \text{ where } n = \mathsf{rd}_{\mathcal{T}}(q) \end{aligned}$$

By this definition, it is obvious that for every q and q', $f_q(q')$ is in a lower class than q (or in Q_0 if $q \in Q_0$). Lemma 6.12 shows that Conditions 1 and 2 of Definition 6.9 are satisfied. It remains to show that $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ is Q_0-looping. If $q \in Q_0$ holds, then there are no existential restrictions in q, and therefore $(q, \emptyset, \ldots, \emptyset)$ is contained in Δ.□

For an automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, \Delta, I)$, the state space Q is exponential in the size of the input $(\mathsf{C}, \mathcal{T})$. The arity k is obviously linear since it is bounded by the number of existential quantifiers appearing in the input; and the number of segments m is also linear since it is bounded by the expanded role depth of C. Thus, by Corollary 6.11, we obtain that satisfiability for $\mathcal{ALC}$ concepts w. r. t. acyclic TBoxes can be decided by a non-deterministic algorithm using space polynomial in the size of the input. Applying the theorem of Savitch (1970) then yields a deterministic complexity class.

Corollary 6.14. Satisfiability of $\mathcal{ALC}$ concepts with respect to acyclic TBoxes is PSPACE-complete.

PSPACE-hardness for this problem follows directly from the known PSPACE-hardness result for $\mathcal{ALC}$ concept satisfiability w. r. t. the empty TBox (Schmidt-Schauß and Smolka, 1991).

6.2 Blocking Automata

Segmentable automata have the advantage that they are comparably easy to handle and that the segmentation of the state space can be rather obvious, as in the case for the role depth in $\mathcal{ALC}$. There are, however, logics that are in PSPACE but do not have the finite tree model property and are also not segmentable. For example, if there is a transitive role r, then the concept $\exists \mathsf{r}.\mathsf{A} \sqcap \forall \mathsf{r}.\exists \mathsf{r}.\mathsf{A}$ does not have a finite tree model. In order to obtain a PSPACE result in this case, the tableau algorithm for $\mathcal{SI}$ (Horrocks et al., 1999) uses *blocking* (see Section 3.2.2) and requires a rather sophisticated blocking condition to ensure that a blocked node is reached after a polynomial number of steps. In this section, we adapt this technique to automata algorithms and extend it to acyclic TBoxes.

6.2.1 An Automata Algorithm for $\mathcal{SI}$ Concept Satisfiability w. r. t. Acyclic and General TBoxes

We begin with formally introducing the DL $\mathcal{SI}$, which extends $\mathcal{ALC}$ with transitive and inverse roles.

Definition 6.15 ($\mathcal{SI}$). Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be sets of concept and role names as in Definition 2.1 and let $\mathsf{N_{R^+}}$, the set of *transitive* role names, be a subset of $\mathsf{N_R}$. For a role name r, r^- is called an *inverse role*. The DL $\mathcal{SI}$ is defined as $\mathcal{ALC}$ (Definition 2.4) with the addition that inverse roles can be used instead of roles in existential and value restrictions.

The semantics of transitive and inverse roles is as defined in Figure 2.1, i. e. transitive role names have to be interpreted by transitive relations, and $(\mathsf{r}^-)^{\mathcal{I}}$ consists of the converses of all pairs in $\mathsf{r}^{\mathcal{I}}$. As usual, in order to avoid having to deal with double inversion as in r^{--}, we define the *inverse of a role* r, $\overline{\mathsf{r}}$, as r^-, if r is a role name, and as s, if r is an inverse role s^-. We use the predicate $\mathit{trans}(\mathsf{r})$ to express that r or $\overline{\mathsf{r}}$ is contained in $\mathsf{N_{R^+}}$. ◇

The $\mathcal{ALC}$ automata algorithm from Section 6.1.1 has to be extended in two ways in order to capture $\mathcal{SI}$. Firstly, for inverse roles, the automaton has to guess in advance which concepts will be required in the current node due to value restrictions in the successors because it does not have the capability of two-way automata to go upward in the tree. However, each transition involves guessing a complete type for every successor, thus the value restrictions that the successors will contain are known in advance. Therefore inverse roles only require a slight modification in the definition of $\mathsf{C}, \mathcal{T}$-compatible. Secondly, for a transitive role r and a value restriction $\forall \mathsf{r}.\mathsf{C}$, the transition relation does not only require C in the label of an r-successor, but also $\forall \mathsf{r}.\mathsf{C}$, which ensures that all indirect r-successors also contain C.

For reasons which will be explained in Section 6.2.3, we will not use a single set of concepts as a node label, but a quadruple $(\Gamma, \Pi, \Omega, \varrho)$, where ϱ is the role that connects the node with its father node, Ω is the entire Hintikka set for the node, $\Gamma \subseteq \Omega$ contains the unique concept D contained in Ω because of an existential restriction $\exists \varrho.\mathsf{D}$ in the father node, and Π contains only those concepts that are contained in Ω because of value restrictions $\forall \varrho \mathsf{E}_i$ in the father node. We will use a special role name λ for nodes that are not connected to the father by a role, i. e. the root node and those (dummy) nodes that are labelled with an empty set of concepts.

Definition 6.16 ($\mathsf{C}, \mathcal{T}$-compatible, Hintikka tree for $\mathcal{SI}$). Let λ be a role name that does not appear in C or $\mathcal{T}$. Then the tuple $((\Gamma_0, \Pi_0, \Omega_0, \varrho_0), (\Gamma_1, \Pi_1, \Omega_1, \varrho_1), \ldots, (\Gamma_k, \Pi_k, \Omega_k, \varrho_k))$ is called *$C, \mathcal{T}$-compatible* if, for all $i \in K$, $\Gamma_i \cup \Pi_i \subseteq \Omega_i$, Ω_i is a $\mathcal{T}$-expanded Hintikka set and, for every existential restriction $\exists \mathsf{r}.\mathsf{D} \in \mathit{sub}(\mathsf{C}, \mathcal{T})$ with $\varphi(\exists \mathsf{r}.\mathsf{D}) = i$, it holds that

- if $\exists \mathsf{r}.\mathsf{D} \in \Omega_0$, then
  1. Γ_i consists of D;
  2. Π_i consists of all concepts E_j for which there is a value restriction $\forall \mathsf{r}.\mathsf{E}_j \in \Omega_0$ and, if r is transitive, additionally $\forall \mathsf{r}.\mathsf{E}_j$;
  3. for every concept $\forall \overline{\mathsf{r}}.\mathsf{F}_j \in \Omega_i$, Ω_0 contains F_j and, if r is transitive, additionally $\forall \overline{\mathsf{r}}.\mathsf{F}_j$;
  4. $\varrho_i = \mathsf{r}$;

- if $\exists \mathsf{r}.\mathsf{D} \notin \Omega_0$, then $\Gamma_i = \Pi_i = \Omega_i = \emptyset$ and $\varrho_i = \lambda$.

The term *Hintikka tree for* C *and* $\mathcal{T}$ is defined accordingly. In a Hintikka tree, we say that a node w is an r-*neighbour* of a node v for a role r if $w = v \cdot \varphi(\exists \mathsf{r}.\mathsf{D})$ for some concept D and $\exists \mathsf{r}.\mathsf{D} \in \Omega(v)$ or if $v = w \cdot \varphi(\exists \overline{\mathsf{r}}.\mathsf{D})$ and $\exists \overline{\mathsf{r}}.\mathsf{D} \in \Omega(w)$. $\diamond$

We will first show that this modification of the definition of $C, \mathcal{T}$-compatible ensures the correct treatment of transitive and inverse roles.

Theorem 6.17. The $\mathcal{SI}$ concept C is satisfiable w. r. t. the general TBox $\mathcal{T}$ iff there exists a Hintikka tree for C and $\mathcal{T}$.

Proof. This proof combines the handling of transitive and inverse roles by Horrocks, Sattler, and Tobies (1998) with the handling of acyclic TBoxes in Section 6.1.1. For a node v with $t(v) = (\Gamma, \Pi, \Omega, \varrho)$, we will refer to the components as $\Gamma(v)$, $\Pi(v)$ etc.

For the "if" direction, we will show how to construct a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ from a Hintikka tree t. Let $\Delta^{\mathcal{I}} = \{v \in K^* \mid t(v) \neq (\emptyset, \emptyset, \emptyset, \lambda)\}$. For a role name $\mathsf{r} \in \mathsf{N_R} \setminus \mathsf{N_{R^+}}$, we define $\mathsf{r}^{\mathcal{I}} = \{(v, w) \mid w \text{ is an } \mathsf{r}\text{-neighbour of } v\}$. If $\mathsf{r} \in \mathsf{N_{R^+}}$, we define $\mathsf{r}^{\mathcal{I}}$ as the transitive closure of this relation.

For a primitive concept name A, we define $\mathsf{A}^{\mathcal{I}} = \{v \in \Delta^{\mathcal{I}} \mid \mathsf{A} \in \Omega(v)\}$. In order to show that this interpretation can be extended to defined concept names and that it interprets complex concepts correctly, we define a weight function $o(\mathsf{C})$ for concept terms C as follows:

- $o(\mathsf{A}) = 1$ for a primitive concept name A;
- $o(\mathsf{A}) = o(\mathsf{C}) + 1$ for a defined concept name $\mathsf{A} \doteq \mathsf{C}$;
- $o(\neg \mathsf{A}) = o(\mathsf{A}) + 1$ for the negation of a (primitive or defined) concept name;
- $o(\mathsf{C} \sqcap \mathsf{D}) = o(\mathsf{C} \sqcup \mathsf{D}) = \max\{o(\mathsf{C}), o(\mathsf{D})\} + 1$;
- $o(\exists \mathsf{r}.\mathsf{C}) = o(\forall \mathsf{r}.\mathsf{C}) = o(\mathsf{C}) + 1$.

Note that o is defined differently from the role depth (Definition 6.2) for the Boolean operators and defined concept names in order to ensure that subconcepts or definitions of a concept have a lower weight than the concept itself. However, o is also well-founded if $\mathcal{T}$ is acyclic. We can now show by induction over the weight of the appearing concepts that $\mathsf{D} \in \Omega(v)$ implies $v \in \mathsf{D}^{\mathcal{I}}$.

- If $\mathsf{A} \in \Omega(v)$ for a primitive concept name A then $v \in \mathsf{A}^{\mathcal{I}}$ holds by definition.
- If $\mathsf{A} \in \Omega(v)$ for a defined concept name $\mathsf{A} \doteq \mathsf{E}$ then $\mathsf{E} \in \Omega(v)$ holds because $\Omega(v)$ is $\mathcal{T}$-expanded. Since $o(\mathsf{E}) < o(\mathsf{A})$, it follows by induction that $v \in \mathsf{E}^{\mathcal{I}}$ holds. Thus we can define $\mathsf{A}^{\mathcal{I}} = \mathsf{E}^{\mathcal{I}}$ and obtain $v \in \mathsf{A}^{\mathcal{I}}$.
- If $\neg \mathsf{A} \in \Omega(v)$ for a negated concept name then $\mathsf{A} \notin \Omega(v)$ holds because $\Omega(v)$ is a Hintikka set. If A is primitive, this implies that $v \notin \mathsf{A}^{\mathcal{I}}$ holds and we are done. If A is a defined concept name and $\mathsf{A} \doteq \mathsf{E}$ then, as in the previous case, $\doteq \neg \mathsf{E} \in \Omega(v)$ holds because $\Omega(v)$ is $\mathcal{T}$-expanded. Again, $o(\doteq \neg \mathsf{E}) < o(\neg \mathsf{A})$ implies $v \in (\doteq \neg \mathsf{E})^{\mathcal{I}}$ by induction and, since $(\doteq \neg \mathsf{E})^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus \mathsf{E}^{\mathcal{I}}$ and $\mathsf{A}^{\mathcal{I}} = \mathsf{E}^{\mathcal{I}}$, it follows that $v \notin \mathsf{A}^{\mathcal{I}}$ holds.

- If $\mathsf{E} \sqcap \mathsf{F} \in \Omega(v)$ then $\Omega(v)$ contains E and F since it is a Hintikka set, and by induction $v \in \mathsf{E}^{\mathcal{I}} \cap \mathsf{F}^{\mathcal{I}}$ holds.
- If $\mathsf{E} \sqcup \mathsf{F} \in \Omega(v)$ then $v \in \mathsf{E}^{\mathcal{I}} \cup \mathsf{F}^{\mathcal{I}}$ follows from an analogous argument.
- If $\exists \mathsf{r}.\mathsf{E} \in \Omega(v)$ for a role name r then, since t is a Hintikka tree, $(v, v \cdot \varphi(\exists \mathsf{r}.\mathsf{E})) \in \mathsf{r}^{\mathcal{I}}$ and $\mathsf{E} \in \Omega(v \cdot \varphi(\exists \mathsf{r}.\mathsf{E}))$ (inverse roles can be treated analogously), thus by induction $v \in (\exists \mathsf{r}.\mathsf{E})^{\mathcal{I}}$ holds.
- If $\forall \mathsf{r}.\mathsf{E} \in \Omega(v)$ for a role r and $(v, w) \in \mathsf{r}^{\mathcal{I}}$ then $(v, w) \in \mathsf{r}^{\mathcal{I}}$ holds either because w is an r-neighbour of v in the Hintikka tree, in which case $\mathsf{E} \in \Omega(w)$ holds by definition of $\mathsf{C}, \mathcal{T}$-compatible, or r is a transitive role and (v, w) is in the transitive closure of the relation defined above. In this case, there exists a sequence of tree nodes $v = v_0, v_1, \ldots, v_{f-1}, v_f = w$ such that for every $i < f$, v_{i+1} is an r-neighbour of v_i. Since $\mathit{trans}(\mathsf{r})$ holds, every node label $t(v_i)$ for $1 \leq i \leq t$ contains $\forall \mathsf{r}.\mathsf{E}$ and E because of the definition of $\mathsf{C}, \mathcal{T}$-compatible, thus it follows by induction that $w \in \mathsf{E}^{\mathcal{I}}$ and $v \in (\forall \mathsf{r}.\mathsf{E})^{\mathcal{I}}$.

For a GCI $\mathsf{E} \sqsubseteq \mathsf{F}$, $\Omega(v)$ contains $\doteq\neg\mathsf{E} \sqcup \mathsf{F}$ for every node v. As $\Omega(v)$ is a Hintikka set, it contains F or $\doteq\neg\mathsf{E}$. If it contains F then, as we have just shown, v belongs to $\mathsf{F}^{\mathcal{I}}$. Otherwise, $\Omega(v)$ contains $\doteq\neg\mathsf{E}$, which implies $v \notin \mathsf{E}^{\mathcal{I}}$ as in the case of negated concept names above. Consequently, every node $v \in \mathsf{E}^{\mathcal{I}}$ is also contained in $\mathsf{F}^{\mathcal{I}}$.

For the "only-if" direction, we show how a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for C w. r. t. $\mathcal{T}$ can be used to define a $\mathsf{C}, \mathcal{T}$-compatible Hintikka tree t with $\mathsf{C} \in \Omega(\varepsilon)$. Let k be the number of existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and φ be a function as in Definition 4.4. We inductively define a function $\vartheta : K^* \to \Delta^{\mathcal{I}} \cup \{\psi\}$ for a new individual ψ such that $\vartheta(v)$ satisfies all concepts in $\Omega(v)$.

Since $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model, there exists an element $d_0 \in \Delta^{\mathcal{I}}$ with $d_0 \in C^{\mathcal{I}}$. So we define $\vartheta(\varepsilon) = d_0$ and set $\Gamma(\varepsilon) = \Pi(\varepsilon) = \emptyset$, $\Omega(\varepsilon) = \{\mathsf{E} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid d_0 \in \mathsf{E}^{\mathcal{I}}\}$, and $\varrho(\varepsilon) = \lambda$. Then we inductively define, for every node v for which ϑ is already defined, the labels of $v \cdot i, 1 \leq i \leq k$, as follows: if $\Omega(v)$ contains the existential restriction $\exists \mathsf{r}.\mathsf{E}$ with $i = \varphi(\exists \mathsf{r}.\mathsf{E})$ then, since $\vartheta(v)$ satisfies $\exists \mathsf{r}.\mathsf{E}$, there exists a $d \in \Delta^{\mathcal{I}}$ with $(\vartheta(v), d) \in \mathsf{r}^{\mathcal{I}}$ and $d \in \mathsf{E}^{\mathcal{I}}$, and thus we define

$$
\begin{array}{rcl}
\vartheta(v \cdot i) & = & d, \\
\Omega(v \cdot i) & = & \{\mathsf{F} \in \mathit{sub}(\mathsf{C}, \mathcal{T}) \mid d \in \mathsf{F}^{\mathcal{I}}\}, \\
\varrho(v \cdot i) & = & \mathsf{r}, \\
\Gamma(v \cdot i) & = & \{\mathsf{E}\}, \text{ and} \\
\Pi(v \cdot i) & & \text{contains for every } \forall \mathsf{r}.\mathsf{F} \in \Omega, \text{ the concept } \mathsf{F} \text{ and,} \\
 & & \text{if } \mathsf{r} \text{ is transitive, additionally } \forall \mathsf{r}.\mathsf{F}.
\end{array}
$$

If $\vartheta(v)$ does not belong to $(\exists \mathsf{r}.\mathsf{E})^{\mathcal{I}}$, we define $\vartheta(v \cdot i) = \psi$ and $(\Gamma(v \cdot i), \Pi(v \cdot i), \Omega(v \cdot i), \varrho(v \cdot i)) = (\emptyset, \emptyset, \emptyset, \lambda)$. It follows by construction that $\Gamma(v \cdot i)$ and $\Pi(v \cdot i)$ are subsets of $\Omega(v \cdot i)$ and that the tuple

$$
\begin{array}{l}
((\Gamma(v), \Pi(v), \Omega(v), \varrho(v)), \\
(\Gamma(v \cdot 1), \Pi(v \cdot 1), \Omega(v \cdot 1), \varrho(v \cdot 1)), \ldots, (\Gamma(v \cdot k), \Pi(v \cdot k), \Omega(v \cdot k), \varrho(v \cdot k)))
\end{array}
$$

is $\mathsf{C}, \mathcal{T}$-compatible. Note that for every $v \in K^*$, $\Omega(v)$ is a Hintikka set since it follows from the fact that $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is a model that if $d \in (\mathsf{E} \sqcup [\sqcap]\mathsf{F})^{\mathcal{I}}$, then $d \in \mathsf{E}^{\mathcal{I}} \cup [\cap]\mathsf{F}^{\mathcal{I}}$, and that $d \in \mathsf{E}^{\mathcal{I}}$ iff $d \notin (\neg\mathsf{E})^{\mathcal{I}}$. □

We will now define an automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ that accepts Hintikka trees for C and $\mathcal{T}$. Adapting the automaton for $\mathcal{ALC}$ from Section 6.1.1 to handle $\mathcal{SI}$ only requires using the quadruples from Definition 6.16 and the corresponding definition of C,$\mathcal{T}$-compatible.

Definition 6.18 (Automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$). For an $\mathcal{SI}$ concept C and a TBox $\mathcal{T}$, let k be the number of existential restrictions in $\mathit{sub}(\mathsf{C}, \mathcal{T})$. Then the looping automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}} = (Q, \Delta, I)$ is defined as follows:

- Q consists of all 4-tuples $(\Gamma, \Pi, \Omega, \varrho)$ such that $\Gamma \cup \Pi \subseteq \Omega \subseteq \mathit{sub}(\mathsf{C}, \mathcal{T})$, Γ is a singleton set, Ω is a $\mathcal{T}$-expanded Hintikka set for C, and ϱ occurs in C or $\mathcal{T}$ or is equal to λ;
- Δ consists of all $\mathsf{C}, \mathcal{T}$-compatible tuples $((\Gamma_0, \Pi_0, \Omega_0, \varrho_0), (\Gamma_1, \Pi_1, \Omega_1, \varrho_1), \ldots, (\Gamma_k, \Pi_k, \Omega_k, \varrho_k))$;
- $I := \{(\emptyset, \emptyset, \Omega, \lambda) \in Q \mid \mathsf{C} \in \Omega\}$. ◇

Soundness and completeness of the automata algorithm as well as the EXPTIME complexity result follow from arguments analogous to those for the $\mathcal{ALC}$ algorithm.

Lemma 6.19. $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ has a successful run iff C is satisfiable w. r. t. $\mathcal{T}$.

Theorem 6.20. Satisfiability in $\mathcal{SI}$ w. r. t. general TBoxes is in EXPTIME.

It is easy to see that the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ for $\mathcal{SI}$ is not segmentable in the sense of Definition 6.6: the role depth does not decrease for concepts involving transitive roles; as in the example from the beginning of this section, a state with $\Gamma = \{\mathsf{A}\}$, $\Pi = \{\forall\mathsf{r}.\exists\mathsf{r}.\mathsf{A}\}$, $\Omega = \{\mathsf{A}, \exists\mathsf{r}.\mathsf{A}, \forall\mathsf{r}.\exists\mathsf{r}.\mathsf{A}\}$ and $\varrho = \mathsf{r}$ can be repeated infinitely often along a path in a run. We therefore develop a framework that is more powerful than the one for segmentable automata and makes use of the property that the number of *different* states on such an infinite path is limited.

6.2.2 The Framework for Blocking Automata

As mentioned in the beginning of Section 6.2, our aim in this section is to restrict the automaton's emptiness test to models of a bounded depth in a similar fashion as in Section 6.1 by adapting the notion of blocking from tableau algorithms. Intuitively, a node v in a completion tree is blocked by a node w if the subtree rooted at w can *replace* the subtree rooted at v. In the automata scheme, the nodes in a successful run are labelled with states, therefore blocking corresponds to the fact that within the successful run, a subtree rooted at a node labelled with the state w can replace a subtree whose root is labelled with v. This is the idea underlying the following definition of $\leftarrow\!\!\prec$-invariant. The expression that an automaton is m-blocking denotes the fact that on every path of every run of this automaton, a blocked state is reached after at most m steps.

Definition 6.21 ($\leftharpoondown$-invariant, m-blocking). Let $\mathcal{A} = (Q, \Delta, I)$ be a looping tree automaton and $\leftharpoondown$ be a binary relation over Q, called the *blocking relation*. If $q \leftharpoondown p$, then we say that q is *blocked* by p. The automaton $\mathcal{A}$ is called $\leftharpoondown$-*invariant* if, for every $q \leftharpoondown p$ with

$$(q_0, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_k) \in \Delta,$$

it holds that

$$(q_0, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_k) \in \Delta.$$

A $\leftharpoondown$-invariant automaton $\mathcal{A}$ is called m-*blocking* if, for every successful run r of $\mathcal{A}$ and every path $v_1, \ldots, v_m$ of length m in r, there are $1 \leq i < j \leq m$ such that $r(v_j) \leftharpoondown r(v_i)$. ◇

Obviously, any looping automaton $\mathcal{A} = (Q, \Delta, I)$ is $=$-invariant (i. e. every node is blocked by itself) and m-blocking for every $m > \#Q$. However, we are interested in automata and blocking relations where blocking occurs earlier than after a linear number of transitions.

As in the case of m-segmentable automata, it is only necessary to consider paths of length m in order to decide the emptiness problem for m-blocking automata. This result is formalised by the next definition and lemma.

Definition 6.22 (Partial run). For a set K as in Definition 4.1, we define $K^{\leq n} := \bigcup_{i=0}^{n} K^i$. A *partial run of depth* m is a mapping $r : K^{\leq m-1} \to Q$ such that $(r(v), r(v \cdot 1), \ldots, r(v \cdot k)) \in \Delta$ for all $v \in K^{\leq m-2}$. It is *successful* if $r(\varepsilon) \in I$. ◇

Lemma 6.23. An m-blocking automaton $\mathcal{A} = (Q, \Delta, I)$ has a successful run iff it has a successful partial run of depth m.

Proof. The "only if" direction is trivial, so only the "if" direction will be proved. For this purpose, we will show how to construct a complete successful run from a partial one by replacing, for every blocked node $v \leftharpoondown w$, the subtree starting at v with the subtree starting at w.

Suppose there is a successful partial run r of depth m. This run will be used to inductively define a function $\beta : K^* \to K^{\leq m}$ as follows:

- $\beta(\varepsilon) := \varepsilon$,
- for a node $v \cdot i$, if there is a predecessor w of $\beta(v) \cdot i$ such that $r(\beta(v) \cdot i) \leftharpoondown r(w)$, then $\beta(v \cdot i) := w$; otherwise $\beta(v \cdot i) := \beta(v) \cdot i$.

The intuitive meaning of $\beta(v) = w$ is "w stands for v"; we will therefore use the labels of w and w's successors in the partial run also for v and v's successors in the complete run. We call the complete run generated in this fashion the *unravelling* of the partial run.

Figure 6.3 shows an example for a partial run of a 3-blocking automaton on a binary tree on the left and its unravelling on the right, where the nodes in the unravelled tree are labelled with their respective beta values. We assume that the nodes 1 and 21 are blocked by ε and that node 22 is blocked by node 2. As an example, we consider the

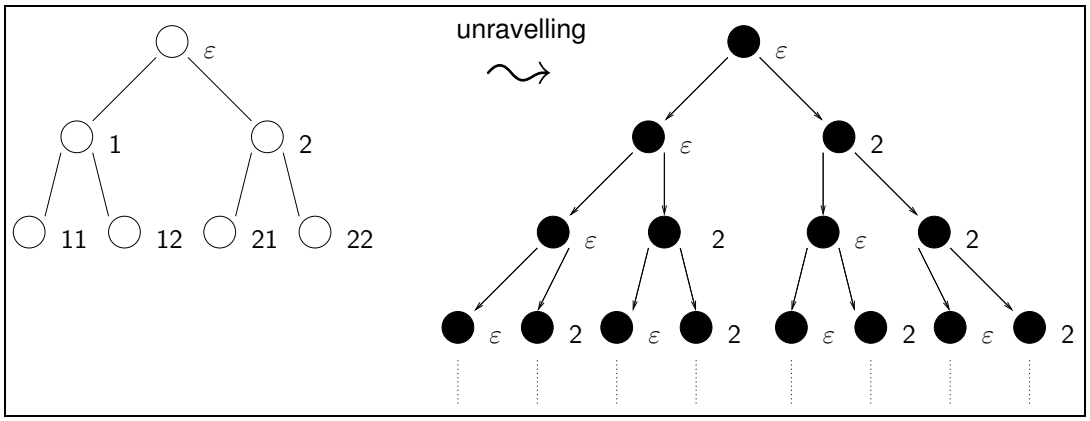

Figure 6.3: Unravelling of a partial run

values of β for the successors of node 21, where $\beta(21) = \varepsilon$. To determine the values of the successors, we have to test if the successors of ε are blocked. For node 211, since $\varepsilon \cdot 1$ is blocked by ε, it turns out that $\beta(211)$ equals ε. On the other hand, for node 212 the corresponding node $\varepsilon \cdot 2$ is not blocked, thus $\beta(212)$ equals 2.

In the following, we will refer to (direct or indirect) successors of blocked nodes as *indirectly blocked*. Notice that the range of β does not contain any blocked or indirectly blocked nodes, since we start with a non-blocked node and, whenever we encounter a blocked node, we replace it and its successors with the blocking one and its successors. Moreover, for every node v with $\beta(v) \neq v$, the depth of v, $|v|$, is larger than $|\beta(v)|$, because β maps a blocked node to a predecessor and the child of a blocked node to a child of the predecessor etc.

We will now show by induction over $|v|$ that the function β is well-defined, more precisely that $|\beta(v)| < m$ for all $v \in K^*$, and that we can use β to construct a successful run s from the successful partial run r by setting $s(v) := r(\beta(v))$ for every node v. For the root, $s(\varepsilon) = r(\varepsilon)$ holds, thus both s and r start with the same label. If, for any node v, the successors of v are not blocked, then the transition $(s(v), s(v{\cdot}1), \ldots, s(v{\cdot}k))$ is contained in Δ because $(r(\beta(v)), r(\beta(v) \cdot 1), \ldots, r(\beta(v) \cdot k))$ is a transition in the run r. In this case, since $\beta(v)$ is not blocked or indirectly blocked, $|\beta(v) \cdot i| < m$ holds for all $1 \leq i \leq k$ because otherwise the path to $\beta(v) \cdot i$ would have a length of at least m without containing a blocked node, in contradiction with the induction hypothesis that the part of s constructed so far is part of a successful run and that neither $\beta(v)$ nor any of its predecessors is blocked.

If any successors of v are blocked, i. e. $r(v \cdot i) \leftarrowtail r(w)$ then $(r(\beta(v)), r(\beta(v) \cdot 1), \ldots, r(\beta(v) \cdot i), \ldots, r(\beta(v) \cdot k)) \in \Delta$ implies $(r(\beta(v)), r(\beta(v) \cdot 1), \ldots, r(\beta(w)), \ldots, r(\beta(v) \cdot k)) \in \Delta$ because of the definition of $\leftarrowtail$-invariance. Hence, $(s(v), s(v \cdot 1), \ldots, s(v \cdot k)) \in \Delta$ holds, and s is a successful run of $\mathcal{A}$. In this case, since w is a predecessor of $\beta(v) \cdot i$ and $|\beta(v)| < m$, it holds that $|w| < m$, and thus $|\beta(v \cdot i)| < m$. Observe that w cannot be blocked itself because $\beta(v)$ is a successor of w or equal to w and the range of β does not contain blocked or indirectly blocked nodes, thus the range of β only contains non-blocked nodes. □

For $k > 1$, the size of a successful partial run of depth m is still exponential in m. However, when checking for the existence of such a run, we can use the same algorithm as for segmentable automata (Figure 6.1) to perform a depth-first traversal of the run while constructing it, using space polynomial in the size of the input. This result will now be shown in a more formal way.

Definition 6.24 (PSPACE on-the-fly construction). Let $\mathfrak{I}$ be set of inputs. A PSPACE *on-the-fly construction* is a construction that yields, for every input $\mathfrak{i} \in \mathfrak{I}$ of size n, an $m_\mathfrak{i}$-blocking automaton $\mathcal{A}_\mathfrak{i} = (Q_\mathfrak{i}, \Delta_\mathfrak{i}, I_\mathfrak{i})$ working on $k_\mathfrak{i}$-ary trees such that there is a polynomial P satisfying the following conditions:

- $m_\mathfrak{i} \leq P(n)$ and $k_\mathfrak{i} \leq P(n)$;
- every element of $Q_\mathfrak{i}$ is of a size bounded by $P(n)$;
- there is a $P(n)$-space bounded non-deterministic algorithm for guessing an element of $I_\mathfrak{i}$; and
- there is a $P(n)$-space bounded non-deterministic algorithm for guessing, on input $q \in Q_\mathfrak{i}$, a transition from $\Delta_\mathfrak{i}$ with first component q.

The algorithms guessing an initial state (a transition starting with q) are assumed to yield the answer "no" if there is no initial state (no such transition). ◇

The following theorem shows that the conditions in Definition 6.24 are sufficient to ensure a PSPACE result. The argument in the proof is very similar to the one in the proof of Theorem 6.8, but since we consider the space consumption of the emptiness test in relation to the *input* $\mathfrak{i}$ rather than the *automaton* $\mathcal{A}_{\mathsf{C},\mathcal{T}}$, we present a formal proof for the theorem.

Theorem 6.25. If the automata $\mathcal{A}_\mathfrak{i}$ are obtained from the inputs $\mathfrak{i} \in \mathfrak{I}$ by a PSPACE on-the-fly construction, then the emptiness problem for $\mathcal{A}_\mathfrak{i}$ can be decided by a deterministic algorithm using space polynomial in the size of $\mathfrak{i}$.

Proof. We will first show by induction that if the algorithm described in Figure 6.1 answers "not empty", then the we can define a successful partial run r from the q_i values used by the algorithm. Since the algorithm answers "not empty", there is an initial transition $(q, q_1, \ldots, q_k)$, so we start with setting $r(\varepsilon) = q$ and $r(i) = q_i$ for all $1 \leq i \leq k$. Suppose now that the algorithm visits a node $v \in K^*$. Then, by induction hypothesis, r is defined for the previously visited nodes. If $\mathit{length}(\mathsf{SN}) < m$ then the algorithm guesses a transition $(r(n), q'_1, \ldots, q'_k)$, which defines the labels of v's children and thus a transition in the run. Otherwise the algorithm has reached depth m, thus we have reached the maximum depth of the partial run.

Conversely, if there is a successful partial run r then it is possible to guess the initial state and initial transition $(r(\varepsilon), r(1), \ldots, r(k))$. By Definition 6.24, the space required for guessing the initial state $r(\varepsilon)$ and the transition from $r(\varepsilon)$ is bounded by $P(n)$. When the algorithm visits one of these initial nodes, they have the same labels as in r. Now suppose the algorithm visits a node v with $r(v) = q$. If the length of v

is smaller than m, then there is a transition on $r, (r(v), r(v \cdot 1), \ldots, r(v \cdot k))$ which the algorithm can guess (using space bounded by $P(n)$) and so it will not return "empty". At any time, the stack SQ contains at most $m_{\mathfrak{i}}$ tuples of $k_{\mathfrak{i}}$ states and SN contains at most $m_{\mathfrak{i}}$ numbers between 0 and $k_{\mathfrak{i}}$. Since $m_{\mathfrak{i}}$, $k_{\mathfrak{i}}$ and the size of each state are bounded by $P(m)$, the space used by these stacks is polynomial in the size of $\mathfrak{i}$.

It follows from Lemma 6.23 that this emptiness test is sound and complete. As in the case of segmentable automata, we obtain the deterministic complexity class from Savitch's theorem. □

6.2.3 Satisfiability in $\mathcal{SI}$ w. r. t. Acyclic TBoxes

It is easy to see that the construction of the automaton $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ from a given $\mathcal{SI}$ concept C and a general TBox $\mathcal{T}$ satisfies all but one of the conditions of a PSPACE on-the-fly construction: the condition that is violated is the one requiring that blocking must occur after a polynomial number of steps. In the case of general TBoxes, this is not surprising since we know that the satisfiability problem is EXPTIME-hard. Unfortunately, this condition is also violated if $\mathcal{T}$ is an acyclic TBox. The reason is similar to the one that made the introduction of *weakly*-segmentable automata necessary: successor states may contain new concepts that are not really required by the definition of $\mathsf{C}, \mathcal{T}$-compatible tuples, but are also not prevented by this definition. In the case of acyclic TBoxes, we can construct a subautomaton that avoids such unnecessary concepts. It has fewer runs than $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ but, similar to the case of weakly-segmentable automata, it does have a successful run whenever $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ has one.

In contrast to segmentable automata, where the notion of *weakly*-segmentable incorporates both the reduction of the automaton (i. e. removal of unnecessary concepts) and the enforcement of the framework's conditions (i. e. the decrease of the role depth), we separate these two aspects in the framework of blocking automata, i. e. we will first construct a subautomaton and then show explicitly that the subautomaton is m-blocking for an appropriate value m. The reason for this is that the definition of segmentable automata requires that *every* transition has to lead to a lower class in the hierarchy, and thus we can meet this demand by removing the inappropriate transitions from the transition relation. In contrast, the notion of blocking makes it necessary to consider paths of length m, i. e. *sequences* of transitions. In other words, we *gradually* approach the level Q_0 of segmentable automata with every transition, but we do not gradually approach a blocked state, and thus it is not practical to satisfy this need by modifying single transitions.

The first step, i. e. the construction of a subautomaton, is captured by the following definition.

Definition 6.26 (Faithful). Let $\mathcal{A} = (Q, \Delta, I)$ be a looping tree automaton on k-ary trees. The family of functions $f_q : Q \to Q^{\mathrm{S}}$ for $q \in Q^{\mathrm{S}}$ is called *faithful* w. r. t. $\mathcal{A}$ if $I \subseteq Q^{\mathrm{S}} \subseteq Q$, and the following two conditions are satisfied for every $q \in Q^{\mathrm{S}}$:

1. if $(q, q_1, \ldots, q_k) \in \Delta$, then $(q, f_q(q_1), \ldots, f_q(q_k)) \in \Delta$;
2. if $(q_0, q_1, \ldots, q_k) \in \Delta$, then $(f_q(q_0), f_q(q_1), \ldots, f_q(q_k)) \in \Delta$.

The *subautomaton* $\mathcal{A}^{\mathrm{S}} = (Q^{\mathrm{S}}, \Delta^{\mathrm{S}}, I)$ of $\mathcal{A}$ *induced by this family* has the transition relation $\Delta^{\mathrm{S}} := \{(q, f_q(q_1), \ldots, f_q(q_k)) \mid (q, q_1, \ldots, q_k) \in \Delta \text{ and } q \in Q^{\mathrm{S}}\}$. ◇

Please note that neither of the two conditions in the definition of "faithful" implies the other one: Condition 1 does not imply Condition 2 because q_0 need not be equal to q. Without Condition 2, we could not be sure that there are still transitions from the direct successor states of q in the reduced automaton $\mathcal{A}^{\mathrm{S}}$. Conversely, Condition 2 does not imply Condition 1 because it is not required that $f_q(q)$ equals q. Intuitively, the range of f_q consists of those states that are still "allowed" after state q has been reached. If we want to ensure that state q is reached only once, it makes sense to define $f_q(q)$ to be different from q.

We can show equivalence of the emptiness of $\mathcal{A}$ and $\mathcal{A}^{\mathrm{S}}$ in the a way that is analogous to the one for segmentable automata (Lemma 6.10).

Lemma 6.27. Let $\mathcal{A}$ be a looping tree automaton and $\mathcal{A}^{\mathrm{S}}$ its subautomaton induced by the faithful family of functions $f_q : Q \to Q^{\mathrm{S}}$ for $q \in Q^{\mathrm{S}}$. Then $\mathcal{A}$ has a successful run iff $\mathcal{A}^{\mathrm{S}}$ has a successful run.

The main idea underlying the definition of the functions f_q is similar to the one for the state hierarchy in Section 6.1.3: if $\mathcal{T}$ is acyclic, then the definition of $\mathsf{C}, \mathcal{T}$-compatibility requires, for a transition $(q, q_1, \ldots, q_k)$ of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$, only the existence of concepts in $q_i = (\Gamma_i, \Pi_i, \Omega_i, \varrho_i)$ that are of a smaller depth than the maximal depth n of concepts in q if ϱ_i is not transitive. However, in the automaton for an $\mathcal{SI}$ concept, we also have to allow for value restrictions of depth n in Π_i if ϱ_i is transitive. All concepts with a higher depth can be removed from the states q_i while still maintaining $\mathsf{C}, \mathcal{T}$-compatibility. By removing these unnecessary concepts, we can ensure that a blocked node is reached after a polynomial number of transitions.

Definition 6.28 (Functions f_q). Let $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ be as in Definition 6.18. For a set S of concepts and a role r, define $S/\mathsf{r} := \{\forall \mathsf{r}.\mathsf{E} \in S \text{ for some } \mathsf{E}\}$.

For two states $q = (\Gamma, \Pi, \Omega, \varrho)$ and $q' = (\Gamma', \Pi', \Omega', \varrho')$ of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ with $\mathsf{rd}_{\mathcal{T}}(\Omega) = n$, we define the function $f_q(q')$ as follows:

- if $\mathsf{rd}_{\mathcal{T}}(\Gamma') \geq \mathsf{rd}_{\mathcal{T}}(\Omega)$, then $f_q(q') := (\emptyset, \emptyset, \emptyset, \lambda)$;
- otherwise, $f_q(q') := (\Gamma', \Pi'', \Omega'', \varrho')$, where
  - $P = \mathsf{sub}_{\leqslant n}(\mathsf{C}, \mathcal{T})/\varrho'$, if $\mathit{trans}(\varrho')$; otherwise $P = \emptyset$;
  - $\Pi'' = \Pi' \cap (\mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}) \cup P)$;
  - $\Omega'' = \Omega' \cap (\mathsf{sub}_{\leqslant n-1}(\mathsf{C}, \mathcal{T}) \cup \Pi'')$. ◇

The definition of Π'' implies that we remove from Π' all concepts with a higher depth than the maximum depth in Ω and we allow for a concept of the same depth as in Ω only if it has the shape $\forall \varrho'.\mathsf{E}$ and ϱ' is transitive. Note that if $\mathcal{T}$ is acyclic, then the set Ω'' defined above is still a $\mathcal{T}$-expanded Hintikka set.

Lemma 6.29. The family of mappings f_q (for states q of $\mathcal{A}_{\mathsf{C},\mathcal{T}}$) introduced in Definition 6.28 is faithful w. r. t. $\mathcal{A}_{\mathsf{C},\mathcal{T}}$.

Proof. We have to show that both conditions of Definition 6.26 are satisfied.

Condition 1. The case that a successor is replaced by $(\emptyset, \emptyset, \emptyset, \lambda)$ cannot occur because in every successor q_i of q, the role depth of Γ_i is strictly smaller than the maximum depth of Ω (see Definition 6.16). Assume that $(q, q_1, \ldots, q_k) \in \Delta$. To prove that $(q, f_q(q_1), \ldots, f_q(q_k))$ is also contained in Δ, we have to show that this transition satisfies the conditions for $\mathsf{C}, \mathcal{T}$-compatibility in Definition 6.16. Number 1 and 4 are obvious. Number 3 holds because we do not remove anything from Ω. Finally, we do not remove any concepts from the Π_i sets because these concepts have a maximum depth of $\mathsf{rd}_\mathcal{T}(\Omega)$ if ϱ_i is transitive, or $\mathsf{rd}_\mathcal{T}(\Omega) - 1$ otherwise. Thus, we only remove concepts from Ω_i, and none of the removed concepts is required.

Condition 2. Let $(q_0, q_1, \ldots, q_k) \in \Delta$. If for some $i > 0$ with $\varphi(\exists \mathsf{r}.\mathsf{D}) = i$, q_i is replaced by $(\emptyset, \emptyset, \emptyset, \lambda)$ then this means that for the concept $\mathsf{D} \in \Gamma_i$, $\mathsf{rd}_\mathcal{T}(\mathsf{D}) \geq \mathsf{rd}_\mathcal{T}(\Omega)$ holds. This implies that the corresponding existential restriction $\exists \mathsf{r}.\mathsf{D}$ in Ω_0 has a strictly larger depth than $\mathsf{rd}_\mathcal{T}(\Omega)$ and therefore will be removed from $f_q(q_0)$. Otherwise we again have to show that the four conditions from Definition 6.16 hold. Number 1 and 4 are again obvious. For Number 3, observe that if $\forall \overline{\mathsf{r}}.\mathsf{F} \in f_q(\Omega_i)$ holds with $\varrho_i = \mathsf{r}$ then $\mathsf{rd}_\mathcal{T}(\forall \overline{\mathsf{r}}.\mathsf{F}) < n$ holds because $\overline{\mathsf{r}} \neq \varrho_i$ and thus neither F nor $\forall \overline{\mathsf{r}}.\mathsf{F}$ will be removed from Ω_0. For Number 2, if $\forall \mathsf{r}.\mathsf{E} \in f_q(\Omega_0)$ then it holds either that $\mathsf{rd}_\mathcal{T}(\forall \mathsf{r}.\mathsf{E}) < n$ or that $\mathsf{rd}_\mathcal{T}(\forall \mathsf{r}.\mathsf{E}) = n$ and $\mathit{trans}(\mathsf{r})$. In the former case, neither E nor $\forall \mathsf{r}.\mathsf{E}$ will be removed from Π_i. In the latter case, $\forall \mathsf{r}.\mathsf{E}$ will not be removed because $\varrho_i = \mathsf{r}$ and $\mathit{trans}(\mathsf{r})$ holds. □

Consequently, $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ has a successful run iff the induced subautomaton $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ has a successful run. It remains to show that there is an appropriate blocking condition that ensures that a blocked state is reached after a polynomial number of transitions.

Lemma 6.30. The construction of $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ from an input consisting of an $\mathcal{SI}$ concept C and an acyclic TBox $\mathcal{T}$ is a PSPACE on-the-fly construction.

Proof. Let $\mathfrak{i} = (\mathsf{C}, \mathcal{T})$ be an input, i. e. an $\mathcal{SI}$ concept and TBox, and $|\mathfrak{i}|$ be the length of $\mathfrak{i}$. The blocking relation $\hookleftarrow_{\mathcal{SI}}$ is defined as follows:

$$(\Gamma_1, \Pi_1, \Omega_1, \varrho_1) \hookleftarrow_{\mathcal{SI}} (\Gamma_2, \Pi_2, \Omega_2, \varrho_2)$$
$$\text{if}$$
$$\Gamma_1 = \Gamma_2, \Pi_1 = \Pi_2, \Omega_1/\overline{\varrho}_1 = \Omega_2/\overline{\varrho}_2, \text{ and } \varrho_1 = \varrho_2.$$

The definition for the Ω component requires an explanation: why is it sufficient to demand equality only for value restrictions involving the role $\overline{\varrho}$? These value restrictions are the ones that enforce certain concepts in the father node. Together with Π and Γ, which consist of the concepts satisfying restrictions *from* the father node, these concepts are the only ones that are relevant for $\mathsf{C}, \mathcal{T}$-compatibility. All other concepts in Ω either result from propositional expansion of these concepts or are superfluous. Hence, states that coincide on Γ, Π, and $\Omega/\overline{\varrho}$ are interchangeable.

We now have to show that there is a polynomial $P(n)$ satisfying the conditions in Definition 6.24.

Every element of $Q_\mathfrak{i}$ is of a size bounded by $P(n)$. Every state label is a subset of $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and therefore has at most $|\mathfrak{i}|$ elements. The size of each of these elements,

in turn, is bounded by $|\mathfrak{i}|$. Thus, the size of each node label is at most quadratic in the size of the input.

There is a $P(n)$-space bounded non-deterministic algorithm for guessing an initial state or successor states for a given state. This is obvious, since the size of every state is bounded by $|\mathfrak{i}|^2$ and all necessary information for the successor states can be obtained from the current state.

The automaton $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ is operating on $k_{\mathfrak{i}}$-ary trees and $m_{\mathfrak{i}}$-blocking, with $m_{\mathfrak{i}} \leq P(n)$ and $k_{\mathfrak{i}} \leq P(n)$. The tree width $k_{\mathfrak{i}}$ is bounded by the number of existential subconcepts of $\mathfrak{i}$ and therefore by $|\mathfrak{i}|$. In order to show a polynomial bound for $m_{\mathfrak{i}}$, we first have to show that $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ is $\leftarrowtail_{\mathcal{SI}}$-invariant. For states $\{q, q_i\} \subseteq Q^{\mathrm{S}}$ with $q = (\Gamma, \Pi, \Omega, \varrho)$ and $q_i = (\Gamma_i, \Pi_i, \Omega_i, \varrho_i)$ let $(q_0, \ldots, q_j, \ldots, q_k)$ be a transition and $q_j \leftarrowtail_{\mathcal{SI}} q_i$. Then the tuple $(q_0, \ldots, q_i, \ldots, q_k)$ is also $\mathsf{C}, \mathcal{T}$-compatible since $\Gamma_j = \Gamma_i$, $\Pi_j = \Pi_i$, $\varrho_j = \varrho_i$ and Ω_j contains the same value restrictions involving $\overline{\varrho}_j$ as Ω_i.

What is the maximum depth of a blocked node in a successful run? Firstly, observe that transitions $(q, q_1, \ldots, q', \ldots, q_k)$ with $q = (\Gamma, \Pi, \Omega, \varrho)$ and $q' = (\Gamma', \Pi', \Omega', \varrho')$ where ϱ' is different from ϱ or not transitive decrease the maximum depth of concepts contained in the state: if ϱ' is not transitive, then $\mathsf{rd}_{\mathcal{T}}(\Omega')$ is smaller than $\mathsf{rd}_{\mathcal{T}}(\Omega)$ by definition. If ϱ' is transitive, but different from ϱ, then Ω' can only have concepts of depth $\mathsf{rd}_{\mathcal{T}}(\Omega)$ if these start with $\forall\varrho'$. Similarly, Ω can only contain concepts of the same depth as its predecessor state if they begin with $\forall\varrho$, which implies that the role depth decreases after two transitions. (This is the key to obtaining a polynomial bound, and it does not hold for general TBoxes, where the GCIs maintain the same role depth in every node.) This depth is bounded by the maximum depth in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and therefore by $|\mathfrak{i}|$, and thus there are only $|\mathfrak{i}|$ such steps possible before depth 0 is reached. After this point, the path will contain a blocked node, since all further nodes are labelled with $(\emptyset, \emptyset, \emptyset, \lambda)$.

Hence, the role depth can only remain the same along a subpath (a subpath is a path which does not need to begin at ε) where every transition involves the same transitive role r. From the definition of Δ, it follows for any subpath with labels $(\Gamma_0, \Pi_0, \Omega_0, \mathsf{r})$, $(\Gamma_1, \Pi_1, \Omega_1, \mathsf{r})$, …, $(\Gamma_\ell, \Pi_\ell, \Omega_\ell, \mathsf{r})$ that $\Pi_i \subseteq \Pi_{i+1}$ holds for all $1 \leq i \leq \ell - 1$, so there are at most $|\mathfrak{i}|$ different sets Π_i possible. By the same argument, it also holds on this subpath that $\Omega_{i+1}/\overline{\mathsf{r}} \subseteq \Omega_i/\overline{\mathsf{r}}$ for all $1 \leq i \leq \ell - 1$. Once again, it is only possible to have a subpath of length $|\mathfrak{i}|$ with different sets. Finally, since Γ_i contains only one concept, there are also at most $|\mathfrak{i}|$ possibilities for this set. In total, every r-subpath of length larger than $|\mathfrak{i}|^3$ must have two nodes $i < j$ such that $\Gamma_j = \Gamma_i$, $\Pi_j = \Pi_i$ and $\Omega_j/\overline{\mathsf{r}} = \Omega_i/\overline{\mathsf{r}}$, and hence $(\Gamma_j, \Pi_j, \Omega_j, \mathsf{r}) \leftarrowtail_{\mathcal{SI}} (\Gamma_i, \Pi_j, \Omega_i, \mathsf{r})$. Thus, an r-subpath for a transitive role r either contains a blocked node or is shorter than $|\mathfrak{i}|^3$ and therefore is followed by a transition with a role different from r, which decreases the maximum depth of concepts contained in Ω. Altogether, we obtain that every path longer than $|\mathfrak{i}|^4$ contains a blocked node.

This concludes the proof that the construction of $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ is a PSPACE on-the-fly construction with $P(n) = n^4$. □

Since we know that C is satisfiable w. r. t. $\mathcal{T}$ iff $\mathcal{A}_{\mathsf{C},\mathcal{T}}$ has a successful run iff $\mathcal{A}^{\mathrm{S}}_{\mathsf{C},\mathcal{T}}$ has a successful run, Theorem 6.25 yields the desired PSPACE upper-bound.

Theorem 6.31. Satisfiability in $\mathcal{SI}$ w. r. t. acyclic TBoxes is in PSPACE.

Thus, we have developed a framework for PSPACE automata that is capable of handling logics that do not have the finite tree model property. For an automaton to fit into this framework, there has to exist a polynomial bound on the length of a path before a state equivalent to a previous one is reached. This framework is based on an adaptation of the blocking technique known from tableau algorithms and requires more involved arguments in the proofs than the segmentable automata framework. It can, however, capture logics that are outside the scope of segmentable automata, as we have shown above for $\mathcal{SI}$, and also all logics within the scope of segmentable automata, as we will see in the next section.

6.2.4 Segmentable Automata as a Special Case

It is possible to regard the blocking automata framework as a generalisation of the segmentable automata framework (and thus segmentable automata as a special case of blocking automata) in the following way: for an m-segmentable and Q_0-looping automaton $\mathcal{A} = (Q, \Delta, I)$, all states $q \in Q_0$ can be the root of a run. Therefore, it is possible to construct an automaton $\mathcal{A}' = (Q', \Delta', I')$ accepting a non-empty language iff $\mathcal{A}$ does, where $Q'_0 = \{q_{\text{acc}}\}$ consists of only one state and every appearance of a state $q \in Q_0$ in Q, Δ and I is replaced with q_{acc} in Q', Δ' and I', respectively.

Then, using equality as the blocking relation $\hookleftarrow$, we can show that the automaton $\mathcal{A}'$ is $(m+1)$-blocking: after at most m transitions, every state belongs to Q'_0, which implies that $\mathcal{A}$ remains in the state q_{acc} forever.[2]

6.3 Chapter Summary

After the negative result of Chapter 5, in which it turned out that it is not possible to transfer the practical efficiency from tableaus to automata by translation, we have established in this chapter that it is possible to improve the (theoretical) complexity results that can be obtained from automata algorithms by employing techniques stemming from tableau algorithms. We have identified two classes of automata for which emptiness can be tested by constructing a run

- interleaved with the automaton itself, i. e. the transitions of the automaton are computed *on-the-fly*;
- in a non-deterministic, top-down, depth-first manner, keeping only one path in memory at a time;
- ensuring that the maximum depth that has to be checked is bounded polynomially by the size of the input.

[2]Note that this result cannot be shown by defining $\hookleftarrow$ in such a way that any two elements of Q_0 block each other because we cannot ensure that every state in Q_0 can be replaced with every other one in every transition.

This way, it is possible to obtain a PSPACE complexity result instead of the EXPTIME result that can be achieved with the "standard" deterministic bottom-up emptiness test.

For the first class, *segmentable automata*, the polynomial bound on the depth results from the fact that every satisfiable input has a model in the shape of a finite tree whose depth is polynomial in the size of the input. By defining a hierarchy over the set of states that consists of a polynomial number of levels, and by restricting that every transition of the automaton implies a descent within the hierarchy, we can ensure that the result of the emptiness test can be determined after a polynomial number of transitions.

The second class, *blocking automata*, is more general. Here, we do not require that every model is a finite tree, but only that it is a *periodical* tree in the sense that every path of a certain length contains two "equivalent" states. It then remains to establish a polynomial bound for this length. This idea is an adaptation of the *blocking* technique known from tableau algorithms, which allows to halt the construction of a tableau after a blocked node has been reached. As an example for an application of this method, we have shown how blocking automata can be used to decide satisfiability of $\mathcal{SI}$ concepts w. r. t. acyclic TBoxes in PSPACE.

In both cases, it was necessary to prevent the introduction of arbitrarily complex concepts in a transition in order to apply these results to automata algorithms for DLs. In the case of segmentable automata, we have shown that *weakly-segmentable* automata satisfy these conditions, i. e. automata for which it is possible to reduce the complexity in a transition. For blocking automata, we used the notion of a *faithful* family of functions in order to define a subautomaton with an appropriate blocking relation.

These results, however, do not imply that a tableau algorithm can be regarded as an implementation of the top-down automata emptiness test: the automaton still guesses a $\mathcal{T}$-expanded Hintikka set for every node during the traversal, i. e. it has to guess in advance a propositionally expanded node label, which additionally can contain arbitrary subconcepts of the input concept and TBox (in the case of weakly-segmentable automata and faithful subautomata, only the depth of the appearing concepts is bounded). In contrast, the tableau algorithm can perform several expansion steps on a single node and thus satisfy the constraints for the node step by step, avoiding non-deterministic decisions involving irrelevant concepts. Therefore, an implementation of our improved automata emptiness test is not likely to perform as well as a tableau algorithm. However, the possibility of transferring the techniques described above from tableau to automata algorithms again emphasises the close relationship between these two approaches.

Chapter 7

Tableau Systems

After establishing the possibility of transferring complexity results from tableau to automata algorithms in Chapter 6, our aim is now to use automata in order to remedy the two main drawbacks of tableau algorithms: firstly, the need to find a blocking condition suitable for the corresponding logic and to prove soundness in the presence of this blocking condition; and secondly, the problem that the "natural" TAs for logics allowing for disjunction are non-deterministic and thus unsuited for proving tight upper complexity bounds for DLs in deterministic complexity classes, e. g. ExpTime. Since the determinism of automata algorithms results from the bottom-up emptiness test which, in turn, is the main reason for the poor performance of AAs in practice, modifying the TA itself in such a way that it runs in deterministic exponential time is not a promising approach: it would sacrifice the main advantage of tableaus.

Instead, our aim in this chapter is to define a *general framework* for ExpTime logics, called *tableau systems*, from which both a practically usable tableau algorithm and a worst-case optimal automata algorithm can be derived, which avoids the need to construct the AA by hand. In order to achieve this, it is necessary to begin with a formalisation of the key properties of TAs such as tableau rules, clash-triggers etc. Afterwards, we show how tableau systems yield automata- and tableau-based algorithms. For the TA, an appropriate blocking condition can be obtained directly from the properties of the tableau system. It is thus possible to prove the correctness of blocking in general, which avoids the need to deal with it for each specific tableau system. Finally, we illustrate the usefulness of tableau systems with two examples, one involving the well-known logic $\mathcal{SHIQ}$ and the other one proving a new ExpTime complexity result for the logic $\mathcal{SHIO}$.

This chapter is based on work that was previously published by Baader, Hladik, Lutz, and Wolter (2003b,c); Hladik and Model (2004); and Hladik (2004).

7.1 The Tableau Systems Framework

We begin with developing a general notion of tableau algorithms. It is in the nature of this endeavour that our formalism will be a rather abstract one. We start with defining the core notion: tableau systems. Intuitively, the purpose of a tableau system is to

capture all the details of a tableau algorithm such as the one for $\mathcal{ALC}$ discussed in Section 3.2.2. The set $\mathfrak{I}$ of inputs used in the following definition can be thought of as consisting of all possible pairs $(\mathsf{C}, \mathcal{T})$ of concepts C and TBoxes $\mathcal{T}$ of the DL under consideration.

Definition 7.1 (Tableau system). Let $\mathfrak{I}$ be a set of *inputs*. A *tableau system for* $\mathfrak{I}$ is a tuple

$$S = (\mathit{NLE}, \mathit{GME}, \mathit{EL}, d, \cdot^S, \mathcal{R}, \mathcal{C}),$$

where NLE, GME, and EL are sets of *node label elements*, *global memory elements*, and *edge labels*, respectively, d is a natural number (the *pattern depth*), and $\cdot^S$ is a function mapping each input $\Gamma \in \mathfrak{I}$ to a tuple

$$\Gamma^S = (\mathit{nle}, \mathit{gme}, \mathit{el}, \mathit{ini})$$

such that

- $\mathit{nle} \subseteq \mathit{NLE}$, $\mathit{gme} \subseteq \mathit{GME}$, and $\mathit{el} \subseteq \mathit{EL}$ are finite;
- ini is a subset of $\wp(\mathit{nle}) \times \wp(\mathit{gme})$, where $\wp(\cdot)$ denotes powerset.

The definitions of $\mathcal{R}$ and $\mathcal{C}$ depend on the notion of an *S-pattern*. Such a pattern is a pair (t, μ) consisting of a a finite labelled tree

$$t = (V, E, n, \ell)$$

of depth at most d with $n : V \to \wp(\mathit{NLE})$ and $\ell : E \to \mathit{EL}$ node and edge labelling functions, and a subset μ of GME. Finally,

- $\mathcal{R}$, the collection of *completion rules*, is a function mapping each possible S-pattern to a finite set of non-empty finite sets of S-patterns; and
- $\mathcal{C}$, the collection of *clash-triggers*, is a set of S-patterns. ◇

In order to illustrate tableau systems, we now define a tableau system $S_{\mathcal{ALC}}$ that describes the $\mathcal{ALC}$ tableau algorithm discussed in Section 3.2.2. As the set of inputs $\mathfrak{I}$ for $S_{\mathcal{ALC}}$, we simply use the set of all $\mathcal{ALC}$-concepts in NNF. Now for the tableau system itself. Let $\mathit{CON}_{\mathcal{ALC}}$ be the set of all $\mathcal{ALC}$ concepts as in Definition 2.4. Intuitively, NLE is the set of elements that may appear in node labels of completion trees, *independently* of the input. In the case of $\mathcal{ALC}$, NLE is thus simply $\mathit{CON}_{\mathcal{ALC}}$. Similarly, EL is the set of edge labels, also independently of the input. In the case of $\mathcal{ALC}$, EL is thus the set of role names $\mathsf{N_R}$. The purpose of the global memory component can be illustrated by the $\mathcal{T}$-rule (Figure 3.3). In contrast to the other rules, which are local in the sense that they are concerned with a single node of the completion tree or a single node and its successor nodes, the $\mathcal{T}$-rule is global: it considers an *arbitrary* node v in the completion tree. The global memory component contains information relevant for such global rules. For the $\mathcal{T}$-rule, it is important to know which concepts $\dot{\neg}\mathsf{D} \sqcup \mathsf{E}$ must be propagated to all nodes because $\mathsf{D} \sqsubseteq \mathsf{E}$ is contained in the TBox. Thus, the global memory component also contains concepts, which means that, in the

case of $\mathcal{ALC}$, GME is also equal to $\mathit{CON}_{\mathcal{ALC}}$. The number d restricts the size of the trees in patterns. We will consider it in more detail when describing the rules and clash-triggers.

The function $\cdot^S$ describes the influence of the input on the form of the constructed completion trees. More precisely, nle fixes the node label elements that may be used in a completion tree for a particular input, and el fixes the edge labels. Similarly, gme fixes the possible elements of the global memory component for a particular input. Finally, ini describes the possible initial node labels of the root of the completion tree as well as the initial value of the global memory component. Note that the initial root label and the initial value of the global memory component are not necessarily unique, but rather there can be many choices—a possible source of (don't-know-)non-determinism that does not show up in the $\mathcal{ALC}$ algorithm.

To illustrate the function $\cdot^S$, let us define it for the tableau system $S_{\mathcal{ALC}}$. For simplicity, we write $\mathit{nle}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T})$ to refer to the first element of the tuple $(\mathsf{C}, \mathcal{T})^{S_{\mathcal{ALC}}}$, $\mathit{gme}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T})$ for the second element, and so forth.

Definition 7.2 (Tableau system $S_{\mathcal{ALC}}$). For an input $(\mathsf{C}, \mathcal{T}) \in \mathit{CON}_{\mathcal{ALC}} \times \wp(\{\mathsf{D} \sqsubseteq \mathsf{E} \mid \mathsf{D}, \mathsf{E} \in \mathit{CON}_{\mathcal{ALC}}\})$, we define

$$\begin{array}{lcl} \mathit{nle}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T}) & = & \mathit{sub}(\mathsf{C}, \mathcal{T}); \\ \mathit{gme}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T}) & = & \{\dot{\neg}\mathsf{D} \sqcup \mathsf{E} \mid \mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}\}; \\ \mathit{el}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T}) & = & \mathit{rol}(\mathsf{C}, \mathcal{T}); \\ \mathit{ini}_{S_{\mathcal{ALC}}}(\mathsf{C}, \mathcal{T}) & = & \{(\{\mathsf{C}\}, \{\dot{\neg}\mathsf{D} \sqcup \mathsf{E} \mid \mathsf{D} \sqsubseteq \mathsf{E} \in \mathcal{T}\})\}. \end{array}$$ ◇

It remains to formalise the completion rules and clash-triggers. First observe that, in the $\mathcal{ALC}$ tableau, every clash-trigger as well as every rule's pre- and postcondition (except for the $\mathcal{T}$-rule) concerns only a single node either alone or together with its successors in the completion tree. For this reason, we can restrict the depth of the trees in patterns to $d = 1$. The global $\mathcal{T}$-rule is handled through the global memory component (see the description of the rules below).

The collection of completion rules $\mathcal{R}$ maps patterns to finite sets of finite sets of patterns. Intuitively, if P is a pattern and $\{P_1, \ldots, P_m\} \in \mathcal{R}(P)$, then this means that a rule of the collection can be applied to all completion trees "matching" the pattern P. For this, the tree part of the pattern must match a subtree of the completion tree, and the global memory component of the pattern must coincide with the global memory component of the completion tree. If a rule matches a completion tree in this sense, then it non-deterministically replaces the matched subtree of the completion tree with a subtree matching the tree part of one of the patterns $P_1, \ldots, P_m$ (we will give a formal definition of this later on). In addition, the global memory component of the completion tree is replaced by the global memory component of the right-hand side pattern. If $\{P_1, \ldots, P_m\} \in \mathcal{R}(P)$, then we will usually write

$$P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$$

to indicate the rule induced by this element of $\mathcal{R}(P)$.

Similar to the application of such a rule, a completion tree contains a clash if this completion tree matches a pattern in $\mathcal{C}$. To illustrate this, let us again consider the

R$\sqcap$ if the root label $n(v_0)$ contains the concept $\mathsf{C} \sqcap \mathsf{D}$ and $\{\mathsf{C}, \mathsf{D}\} \not\subseteq n(v_0)$,
then $\mathcal{R}(P)$ contains the singleton set $\{((V, E, n', \ell), \mu)\}$,
where $n'(v) = n(v)$ for all $v \in V \setminus \{v_0\}$ and $n'(v_0) = n(v_0) \cup \{\mathsf{C}, \mathsf{D}\}$

R$\sqcup$ if the root label $n(v_0)$ contains the concept $\mathsf{C} \sqcup \mathsf{D}$ and $\{\mathsf{C}, \mathsf{D}\} \cap n(v_0) = \emptyset$,
then $\mathcal{R}(P)$ contains the set $\{((V, E, n', \ell), \mu), ((V, E, n'', \ell), \mu)\}$,
where $n'(v) = n''(v) = n(v)$ for all $v \in V \setminus \{v_0\}$,
$n'(v_0) = n(v_0) \cup \{\mathsf{C}\}$ and $n''(v_0) = n(v_0) \cup \{\mathsf{D}\}$

R$\exists$ if the root label $n(v_0)$ contains the concept $\exists \mathsf{r}.\mathsf{C}$,
$u_1, \ldots, u_m$ are all the children of v_0 with $\ell(v_0, u_i) = \mathsf{r}$,
and $\mathsf{C} \notin n(u_i)$ for all $i, 1 \leq i \leq m$,
then $\mathcal{R}(P)$ contains the set $\{P_0, P_1, \ldots, P_m\}$, where

- $P_0 = ((V_0, E_0, n_0, \ell_0), \mu)$, where u_0 is a node not contained in V,
  $V_0 = V \cup \{u_0\}$, $E' = E \cup \{(v_0, u_0)\}$,
  $n_0 = n \cup \{u_0 \mapsto \{\mathsf{C}\}\}$, $\ell' = \ell \cup \{(v_0, u_0) \mapsto \mathsf{r}\}$, and
- for $i = 1, \ldots, m$, $P_i = ((V, E, n_i, \ell), \mu)$,
  where $n_i(v) = n(v)$ for all $v \in V \setminus \{u_i\}$ and $n_i(u_i) = n(u_i) \cup \{\mathsf{C}\}$

R$\forall$ if $n(v_0)$ contains the concept $\forall \mathsf{r}.\mathsf{C}$, $\ell(v_0, v_1) = \mathsf{r}$ for some $v_1 \in V$, and $\mathsf{C} \notin n(v_1)$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(v) = n(v)$ for all $v \in V \setminus \{v_1\}$ and $n'(v_1) = n(v_1) \cup \{\mathsf{C}\}$

R$\mathcal{T}$ if μ contains the concept C and $\mathsf{C} \notin n(v_0)$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(v) = n(v)$ for all $v \in V \setminus \{v_0\}$ and $n'(v_0) = n(v_0) \cup \{\mathsf{C}\}$

Figure 7.1: Rules of the $\mathcal{ALC}$ tableau system

case of $\mathcal{ALC}$. Here the set of clash-triggers $\mathcal{C}$ consists of all patterns whose tree has a root label containing both A and $\neg\mathsf{A}$ for some concept name A. The effect of this is that a completion tree contains a clash iff one of its node labels contains A and $\neg\mathsf{A}$ for some concept name A.

For each pattern $P = (t, \mu)$ with $t = (V, E, n, \ell)$ being a tree of depth at most 1 with root v_0, $\mathcal{R}(P)$ is the smallest set of finite sets of patterns such that the conditions in Figure 7.1 hold. The collection of completion rules is defined by a straightforward translation of the rules in Figures 3.2 and 3.3 with one exception, namely the treatment of existential restrictions. The rule in Figure 3.2 is deterministic: it always generates a *new* r-successor of the given node. In contrast, the rule handling existential restrictions introduced above (don't-know-)non-deterministically chooses between generating a new successor or re-using one of the old ones. Basically, this additional non-determinism is the price we have to pay for having a very general framework. The reason why one can always create a new individual when treating existential restrictions in $\mathcal{ALC}$ is that $\mathcal{ALC}$ is invariant under bisimulation (Blackburn et al., 2001),

and thus one can duplicate successors in models without changing validity. We could have tailored our framework such that the deterministic rule for $\mathcal{ALC}$ can be used, essentially by dropping the restriction that the function π in Definition 7.6 below, which maps nodes from the pattern P to nodes in the completion tree T, has to be injective. However, in this case we would have restricted the applicability of tableau systems to DLs invariant under bisimulation, a property that is violated by other DLs such as those providing for number restrictions (see Section 7.4 for an example).

Let us now continue with the general definitions. Tableau systems are a rather general notion. In fact, as described until now they are too general to be useful for our purposes. For example, tableau algorithms described by such tableau systems need not be monotonic: completion rules could repeatedly (even indefinitely) add and remove the same piece of information. To prevent such pathologic behaviour, we now formulate a number of conditions that "well-behaved" tableau systems are supposed to satisfy. For the following definitions, fix a set of inputs $\mathfrak{I}$ and a tableau system $S = (\mathsf{NLE}, \mathsf{GME}, \mathsf{EL}, d, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$.

Before we can define admissibility of tableau systems, we must introduce an "inclusion relation" between patterns.

Definition 7.3 (Relations between patterns). Let $P = (t, \mu)$ and $P' = (t', \mu')$ with $t = (V, E, n, \ell)$ and $t' = (V', E', n', \ell')$ be S-patterns. We write $P \precsim P'$ if the following conditions are satisfied: $\mu \subseteq \mu'$ and there is an injection $\pi : V \to V'$ that maps the root of t to the root of t' and satisfies the following conditions:

- for all $x \in V$, we have $n(x) \subseteq n'(\pi(x))$; and
- for all $x, y \in V$, $(x, y) \in E$ implies $(\pi(x), \pi(y)) \in E'$ and $\ell(x, y) = \ell'(\pi(x), \pi(y))$. ◇

If π is the identity on V (and thus $V \subseteq V'$), then we write $P \preceq P'$ (and $P \prec P'$ if, additionally, $P \neq P'$). If $\mu = \mu'$, π is a bijection, and $n(x) = n'(\pi(x))$ for all $x \in V$, then we write $P \sim P'$. To make the injection (bijection) π explicit, we sometimes write $P \precsim_\pi P'$ ($P \sim_\pi P'$).

For an input $\mathfrak{i} \in \mathfrak{I}$ we say that $P = (t, \mu)$ is a *pattern for* $\mathfrak{i}$ iff μ is a subset of $\mathsf{gme}_S(\mathfrak{i})$, the labels of all nodes in t are subsets of $\mathsf{nle}_S(\mathfrak{i})$, and the labels of all edges in t belong to $\mathsf{el}_S(\mathfrak{i})$. The pattern P is *saturated* iff $\mathcal{R}(P) = \emptyset$.

Definition 7.4 (Admissible). The tableau system S is called *admissible* iff it satisfies, for all S-patterns P and P', the following conditions:

1. If $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, then $P \prec P_i$ for all i, with $1 \leq i \leq m$.
2. If $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, P' is saturated, and $P \precsim P'$, then there exists an i, with $1 \leq i \leq m$, such that $P_i \precsim P'$.
3. For all inputs $\mathfrak{i} \in \mathfrak{I}$, if P is a pattern for $\mathfrak{i}$ and $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, then the patterns P_i are patterns for $\mathfrak{i}$.
4. If $P \in \mathcal{C}$ and $P \precsim P'$, then $P' \in \mathcal{C}$. ◇

It is in order to discuss the intuition underlying the above conditions. Condition 1 says that rule application always adds nodes, elements of node labels, or elements of the global memory component. Condition 2 can be understood as follows. Assume that a (non-deterministic) rule is applicable to P and that P' is a "super-pattern" of P that is saturated (i. e. all applicable rules have already been applied). Then the non-deterministic rule can be applied in such a way that the obtained new pattern is still a sub-pattern of P'. Intuitively, this condition can be used to reach P' from P by repeated rule application. Condition 3 says that, by applying completion rules for some input i, we stay within the limits given by the values of the $\cdot^S$ function. Condition 4 states that applicability of clash-triggers is monotonic, i. e. if a pattern triggers a clash, all its "super-patterns" also trigger a clash.

It is easy to see that these conditions are satisfied by the tableau system $S_{\mathcal{ALC}}$ for $\mathcal{ALC}$. For Condition 1, this is obvious since the rules only add nodes, elements of node labels, or elements of the global memory component, but never remove them. Condition 3 holds since rules only add subconcepts of existing concepts to the node label or the global memory component. Condition 4 is also clear: if $P = (t, \mu)$ and the label of the root of t contains A and $\neg\mathsf{A}$, then the label of the root of the tree of every super-pattern of P also contains A and $\neg\mathsf{A}$.

The most interesting condition is Condition 2. We illustrate it by considering the treatment of disjunction and of existential restrictions in $S_{\mathcal{ALC}}$. Firstly, assume that $P \to_{\mathcal{R}} \{P_1, P_2\}$ where the root label of the tree t of P contains $\mathsf{C} \sqcup \mathsf{D}$ and the root labels of the trees of P_1 and P_2 are obtained from the root label of t by respectively adding C and D. If $P \precsim P'$, then the root label of the tree of P' also contains $\mathsf{C} \sqcup \mathsf{D}$. If, in addition, P' is saturated, then this root label already contains C or D. Thus, $P' \precsim P_1$ holds in the first case and $P' \precsim P_2$ holds in the second one.

Secondly, consider the rules handling existential restrictions. Let $P \precsim P'$, and assume that the root label of the tree t of P contains the existential restriction $\exists\mathsf{r}.\mathsf{C}$ and that the root of t has m r-successors $u_1, \ldots, u_m$. Then the existential restriction $\exists\mathsf{r}.\mathsf{C}$ induces the rule $P \to_{\mathcal{R}} \{P_0, \ldots, P_m\}$ where the patterns $P_0, \ldots, P_m$ are as defined above. If, in addition, P' is saturated, then the root of its tree has an r-successor whose label contains C. If this is a "new" r-successor (i. e. one not in the range of the injection π that ensures $P \precsim P'$), then $P_0 \precsim P'$.[1] Otherwise, there is an r-successor u_i of the root of t such that the label of $\pi(u_i)$ in the tree of P' contains C. In this case, $P_i \precsim P'$ holds.

We now introduce S-trees, the abstract counterpart of completion trees, and define what it means for a pattern to match into an S-tree.

Definition 7.5 (S-tree, matching). An *S-tree* is a pair $T = (t, \mu)$ where $\mu \subseteq \mathit{GME}$ and $t = (V, E, n, \ell)$ is a labelled tree with finite out-degree, a countable set of nodes V, and the node and edge labelling functions $n : V \to \wp(\mathit{NLE})$ and $\ell : E \to \mathit{EL}$.

Any node $x \in V$ defines a pattern $T|_x$, the *d-neighbourhood* of x in T, as follows: $T|_x := ((V', E', n', \ell'), \mu)$ where

[1] This shows that we cannot replace $\precsim$ by $\preceq$ in the statement of Condition 2. In fact, we cannot be sure that the new successor introduced in P_0 has the same name as the new successor in P'.

- $V' = \{x\} \cup \{y \in V \mid \text{there is a path from } x \text{ to } y \text{ of length at most } d \text{ in } t\}$;
- E', n', ℓ' are the restrictions of E, n, ℓ to V'.

The tree (V', E', n', ℓ') of $T|_x$ is denoted by $t|_x$. If P is an arbitrary S-pattern and $x \in V$, then we say that P *matches* x in T iff $P \sim T|_x$ (see Definition 7.3). $\diamond$

For the tableau system for $\mathcal{ALC}$ introduced above, $S_{\mathcal{ALC}}$-trees are basically the completion trees defined in Section 3.2. The only difference is that $S_{\mathcal{ALC}}$-trees have an additional global memory component μ.

Later on, we need sub-tree relations between S-trees in analogy to the inclusion relations "$\precsim$" and "$\preceq$" between patterns introduced in Definition 7.3. These relations are defined on trees exactly as for patterns, and we also use the same relation symbols for them.

We are now ready to describe rule application on an abstract level. Intuitively, the rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ can be applied to the node x in the S-tree T if $P \sim T|_x$, and its application yields the new tree T', which is obtained from T by adding new nodes to $T|_x$ and/or extending labels of nodes from $T|_x$ and/or extending the global memory component, as indicated by some P_i. This intuition is formalised in the following definition.

Definition 7.6 (Rule application). Let S be an admissible tableau system, $T = (t, \mu)$ be an S-tree, and $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ be a rule of S. The S-tree $T' = (t', \mu')$ is obtained from T by *application* of this rule to a node x of t iff the following conditions hold:

1. $P \sim_\pi T|_x$ for some bijection π.
2. There is an $i, 1 \leq i \leq m$ such that T' is obtained from T by replacing $T|_x$ by P_i. To be more precise, let $t = (V, E, n, \ell)$, $P = (t_0, \mu_0)$ where $t_0 = (V_0, E_0, n_0, \ell_0)$, and $P_i = (t_i, \mu_i)$ where $t_i = (V_i, E_i, n_i, \ell_i)$, and assume (without loss of generality) that $V \cap V_i = \emptyset$. Let π' be the extension of π to V_i that is the identity on $V_i \setminus V_0$. Then $\mu' = \mu_i$ and $t' = (V', E', n', \ell')$, where
   (a) $V' = V \cup (V_i \setminus V_0)$;
   (b) $E' = E \cup \{(\pi'(y), \pi'(z)) \mid (y, z) \in E_i\}$;
   (c) $n'(y') = n(y')$ if $y' \notin \mathsf{ran}(\pi')$ and $n'(y') = n_i(y)$ if $y' = \pi'(y)$ for some $y \in V_i$;
   (d) $\ell'(y, z) = \ell(y, z)$ for all $(y, z) \in E$, and
   $\ell'(y', z') = \ell_i(y, z)$ if $y' = \pi'(y)$, $z' = \pi'(z)$, and $(y, z) \in E_i$. $\diamond$

For a fixed rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$, a fixed choice of P_i, and a fixed node x in T, the result of the rule application is unique. It is easy to check that, in the case of $S_{\mathcal{ALC}}$, rule application as defined above captures precisely the intuitive understanding of rule application employed in Section 3.2.

To finish our abstract definition of tableau algorithms, we need some way to describe the set of S-trees that can be obtained by starting with an initial S-tree for an input $\mathfrak{i}$, and then repeatedly applying completion rules. This leads to the notion of S-trees for $\mathfrak{i}$.

Definition 7.7 (S-tree for $\mathfrak{i}$). Let S be an admissible tableau system, and let $\mathfrak{i}$ be an input for S. The set of *S-trees for $\mathfrak{i}$* is the smallest set of S-trees such that

1. All *initial S-trees for $\mathfrak{i}$* belong to this set, where an initial S-tree for $\mathfrak{i}$ is of the form $((\{v_0\}, \emptyset, \{v_0 \mapsto \Lambda\}, \emptyset), \mu)$, where v_0 is a node and $(\Lambda, \mu) \in \mathit{ini}_S(\mathfrak{i})$.
2. If T is an S-tree for $\mathfrak{i}$ and T' can be obtained from T by the application of a completion rule, then T' is an S-tree for $\mathfrak{i}$.
3. If $T_0, T_1, \ldots$ is an infinite sequence of S-trees for $\mathfrak{i}$ with $T_i = ((V_i, E_i, n_i, \ell_i), \mu_i)$ such that
   (a) T_0 is an initial S-tree for $\mathfrak{i}$ and
   (b) for all $i \geq 0$, T_{i+1} can be obtained from T_i by the application of a completion rule,

   then the S-tree $T^\omega = ((V, E, n, \ell), \mu)$ is also an S-tree for $\mathfrak{i}$, where
   - $V = \bigcup_{i \geq 0} V_i$,
   - $E = \bigcup_{i \geq 0} E_i$,
   - $n = \bigcup_{i \geq 0} n_i$,
   - $\ell = \bigcup_{i \geq 0} \ell_i$, and
   - $\mu = \bigcup_{i \geq 0} \mu_i$. ◇

Rule application may terminate after finitely many steps or continue forever. The last case of Definition 7.7 deals with such infinite sequences of rule applications. The S-tree T^ω can be viewed as the limit of the sequence $T_0, T_1, \ldots$ of S-trees. This limit exists since admissibility of S implies that rule application is monotonic w. r. t. the sub-tree relationship "$\preceq$", i. e. it extends S-trees by new nodes or by additional elements in node labels, but it never removes nodes or elements of node labels.

Let us now define when an S-tree is saturated and clash-free. Saturatedness says that no completion rule is applicable to the S-tree, and an S-tree is clash-free if no clash-trigger can be applied to any of its nodes.

Definition 7.8 (Saturated, clash-free). Let S be an admissible tableau system. We say that the S-tree T is

- *saturated* if, for every node x in T and every pattern P, $P \sim T|_x$ implies $\mathcal{R}(P) = \emptyset$;
- *clash-free* if, for every node x in T and every $P \in \mathcal{C}$, we have $P \not\sim T|_x$. ◇

Finally, we define soundness and completeness of tableau systems w. r. t. a certain property of its set of inputs. If the inputs are concepts (pairs consisting of a concept and a TBox), the property is usually satisfiability of the concept (w. r. t. the TBox).

Definition 7.9 (Sound, complete). Let $\mathcal{P} \subseteq \mathfrak{I}$ be a property. The tableau system S is called

- *sound for* $\mathcal{P}$ iff, for any $\mathfrak{i} \in \mathfrak{I}$, the existence of a saturated and clash-free S-tree for $\mathfrak{i}$ implies that $\mathfrak{i} \in \mathcal{P}$;
- *complete for* $\mathcal{P}$ iff, for any $\mathfrak{i} \in \mathcal{P}$, there exists a saturated and clash-free S-tree for $\mathfrak{i}$. ◇

It should be noted that the algorithmic treatment of tableau systems requires a stronger notion of completeness: an additional condition is needed to ensure that the out-degree of S-trees is appropriately bounded (see Definitions 7.10 and 7.22 below).

Taking into account the soundness and completeness results shown in Theorem 3.6 for the $\mathcal{ALC}$ tableau algorithm described in Figures 3.2 and 3.3, it is straightforward to check that the tableau system $S_{\mathcal{ALC}}$ is sound and complete w. r. t. satisfiability of concepts. Note, in particular, that saturated S-trees for an input $\mathfrak{i}$ are precisely those S-trees for $\mathfrak{i}$ that can be obtained by exhaustive or infinite and *fair* rule application.

7.2 ExpTime Automata Algorithms from Tableau Systems

In this section, we define the class of "ExpTime-admissible" tableau systems. If such a tableau system is sound and complete for a property $\mathcal{P}$, then it gives rise to an ExpTime algorithm for deciding $\mathcal{P}$.[2] In the case where $\mathcal{P}$ is satisfiability of description logic concepts (w. r. t. a TBox), this means that the mere existence of an ExpTime-admissible tableau system for the DL implies an ExpTime upper-bound for concept satisfiability (w. r. t. TBoxes) in this DL. The ExpTime upper-bound is shown via a translation of the inputs of the ExpTime-admissible tableau system into a special kind of non-deterministic tree automata. Since these automata operate on infinite trees, ExpTime-admissible tableau systems need *not* deal with the issue of termination. Indeed, non-terminating tableau algorithms such as the one for $\mathcal{ALC}$ with general TBoxes introduced in Definition 7.2 may yield ExpTime-admissible tableau systems.

Throughout this section, we consider a fixed set of inputs $\mathfrak{I}$ and a fixed tableau system $S = (\mathsf{NLE}, \mathsf{GME}, \mathsf{EL}, d, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$, which is sound and complete w. r. t. some property $\mathcal{P}$. As usual, the exponential upper-bound of deciding $\mathcal{P}$ is assumed to be in the "size" of the input $\mathfrak{i} \in \mathfrak{I}$. Thus, we assume that the set of inputs is equipped with a size function, which assigns to an input $\mathfrak{i} \in \mathfrak{I}$ a natural number, its size $|\mathfrak{i}|$.

[2] More precisely, we must demand a slightly stronger version of completeness, as introduced in Definition 7.10 below.

7.2.1 Basic Notions

Recall that a tableau system S is sound and complete for a property $\mathcal{P}$ if, for any input $\mathfrak{i}$, we have $\mathfrak{i} \in \mathcal{P}$ iff there exists a (potentially infinite) saturated and clash-free S-tree for $\mathfrak{i}$. The fundamental idea for obtaining an ExpTime upper-bound for deciding $\mathcal{P}$ is to use non-deterministic tree automata as in Section 4.3 to check for the existence of a clash-free and saturated S-tree for a given input $\mathfrak{i}$. Since these automata work on trees of some fixed out-degree, this approach only works if the (size of the) input determines such a fixed out-degree for the S-trees to be considered. This motivates the following definition.

Definition 7.10 (p-complete). Let p be a polynomial. The tableau system S is called *p-complete for $\mathcal{P}$* iff, for any $\mathfrak{i} \in \mathcal{P}$, there exists a saturated and clash-free S-tree for $\mathfrak{i}$ with out-degree bounded by $p(|\mathfrak{i}|)$. ◇

Throughout this section, we assume that there exists a polynomial p such that the fixed tableau system S is p-complete w. r. t. the property $\mathcal{P}$ under consideration. The tableau system $S_{\mathcal{ALC}}$ from Definition 7.2 is easily proved to be i-complete, with i being the identity function on the natural numbers: using the formulation of the rules, it is easily seen that the out-degree of every $S_{\mathcal{ALC}}$-tree for the input C is bounded by the number of concepts of the form $\exists \mathsf{r}.\mathsf{D}$ in $\mathit{sub}(\mathsf{C}, \mathcal{T})$ and thus also by the length of the concept C and the TBox $\mathcal{T}$.

It should be noted that most standard description logic tableau algorithms also exploit p-completeness of the underlying logic: although this is not made explicit in the formulation of the algorithm itself, it is usually one of the central arguments in termination proofs, as for Theorems 3.2 and 3.6.[3] The intuition that p-completeness is *not* an artefact of using an automata approach is supported by the fact that a similar strengthening of the notion of completeness is needed in Section 7.3, where we construct tableau algorithms from tableau systems.

To ensure that the automaton $\mathcal{A}_\mathfrak{i}$ can be computed and tested for emptiness in exponential time, we require the function $\cdot^S$ of the tableau system S and the rules of S to exhibit an "acceptable" computational behaviour. This is captured by the following definition, where we assume that all patterns are appropriately encoded in some finite alphabet, and thus can be the input for a decision procedure. The *size of a pattern P* is the sum of the sizes of its global memory component and its node and edge labels, where the size of a node label (global memory component) is the sum of the sizes of its node label elements (global memory elements).

Definition 7.11 (ExpTime-admissible). The tableau system S is called ExpTime-*admissible* iff all of the following conditions are satisfied:

1. S is admissible (see Definition 7.4);
2. $\mathit{ini}_S(\mathfrak{i})$ and $\mathit{el}_S(\mathfrak{i})$ can be computed in time exponential in $|\mathfrak{i}|$, and the size of each edge label in $\mathit{el}_S(\mathfrak{i})$ is polynomial in $|\mathfrak{i}|$;

[3] An exception are algorithms that treat qualifying number restrictions with numbers coded in binary in a naive way (Hollunder and Baader, 1991; Tobies, 1999).

3. the cardinality of $\mathit{nle}_S(\mathfrak{i})$ and the size of each node label element in $\mathit{nle}_S(\mathfrak{i})$ is polynomial in $|\mathfrak{i}|$, and $\mathit{nle}_S(\mathfrak{i})$ can be computed in time exponential in $|\mathfrak{i}|$;
4. the cardinality of $\mathit{gme}_S(\mathfrak{i})$ and the size of each global memory element in $\mathit{gme}_S(\mathfrak{i})$ is polynomial in $|\mathfrak{i}|$, and $\mathit{gme}_S(\mathfrak{i})$ can be computed in time exponential in $|\mathfrak{i}|$;
5. for each pattern P it can be checked in time exponential in the size of P whether, for all patterns P', $P' \sim P$ implies $\mathcal{R}(P') = \emptyset$;
6. for each pattern P it can be checked in time exponential in the size of P whether there is a clash-trigger $P' \in \mathcal{C}$ such that $P' \sim P$. ◇

Note that Condition 2 of ExpTime-admissibility implies that, for each $\mathfrak{i} \in \mathfrak{I}$, the cardinality of the sets $\mathit{ini}_S(\mathfrak{i})$ and $\mathit{el}_S(\mathfrak{i})$ are at most exponential in $|\mathfrak{i}|$. The cardinality of the set of node label elements $\mathit{nle}_S(\mathfrak{i})$ is explicitly required (in Condition 3) to be polynomial. For the actual set of possible node labels (which are sets of node label elements), this yields an exponential upper-bound on its cardinality, but the size of each single node label is polynomial in $|\mathfrak{i}|$. The same is true for the global memory component (Condition 4). ExpTime-admissibility ensures that the size of each d-neighbourhood $T|_x$ is polynomial in $|\mathfrak{i}|$ since

- p-completeness implies that we consider only S-trees T of out-degree bounded by $p(|\mathfrak{i}|)$, and thus the out-degree of each d-neighbourhood is polynomial in $|\mathfrak{i}|$;
- d-neighbourhoods have constant depth d (not depending on the input);
- the sizes of the global memory component and of edge and node labels are polynomial in $|\mathfrak{i}|$.

Thus, the fifth condition ensures that the saturatedness condition can be checked in time exponential in $|\mathfrak{i}|$ for a given neighbourhood $T|_x$ of T. The sixth condition yields the same for clash-freeness.

Most of the standard tableau algorithms for ExpTime-complete DLs trivially satisfy the conditions of ExpTime-admissibility. For example, it is easy to show that the tableau system $S_{\mathcal{ALC}}$ defined in Section 7.1 is ExpTime-admissible. We have already shown admissibility of $S_{\mathcal{ALC}}$, and Condition 2, 3, and 4 are immediate consequences of the definitions of $\mathit{ini}_{S_{\mathcal{ALC}}}$, $\mathit{nle}_{S_{\mathcal{ALC}}}$, $\mathit{gme}_{S_{\mathcal{ALC}}}$, and $\mathit{el}_{S_{\mathcal{ALC}}}$. To see that Conditions 5 and 6 are satisfied as well, first note that the definition of the rules and clash-triggers in $S_{\mathcal{ALC}}$ is invariant under isomorphism of patterns. For this reason, the decision problem in Condition 5 reduces to checking whether a given pattern P is saturated (see the definition of this notion below Definition 7.3), and the decision problem in Condition 6 reduces to checking whether a given pattern is a clash-trigger. As an example, we consider the rule handling existential restrictions. Let $P = ((V, E, n, \ell), \mu)$ be a pattern whose tree has root v_0, and assume that $\exists \mathsf{r}.\mathsf{C} \in n(v_0)$. This existential restriction contributes a set of patterns to $\mathcal{R}(P)$ iff $\mathsf{C} \notin n(u)$ for all r-successors u of v_0. Obviously, this can be checked in time polynomial in the size of the pattern.

The remainder of the present section is concerned with converting ExpTime-admissible tableau systems into automata algorithms, as outlined above. The major

challenge is to bring together the different paradigms underlying tableau and automata algorithms, i. e. to transform the "constructive" tableau rules from the Definitions 7.6 and 7.7 into "accepting" automata transitions. Due to these different perspectives, it is not straightforward to construct automata that directly check for the existence of S-trees for an input $\mathfrak{i}$. To overcome this problem, we first introduce the (less constructive) notion of S-trees *compatible with* $\mathfrak{i}$, and investigate the relationship of this notion to S-trees *for* $\mathfrak{i}$, as introduced in Definition 7.7.

Definition 7.12 (S-tree compatible with $\mathfrak{i}$). Let $\mathfrak{i}$ be an input and $T = ((V, E, n, \ell), \mu)$ an S-tree with root v_0. Then T is *compatible with* $\mathfrak{i}$ iff it satisfies the following conditions:

1. $\mu \subseteq \wp(\mathit{gme}_S(\mathfrak{i}))$;
2. $n(x) \subseteq \wp(\mathit{nle}_S(\mathfrak{i}))$ for each $x \in V$;
3. $\ell(x, y) \in \mathit{el}_S(\mathfrak{i})$ for each $(x, y) \in E$;
4. there exists $(\Lambda, \nu) \in \mathit{ini}_S(\mathfrak{i})$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$; and
5. the out-degree of T is bounded by $p(|\mathfrak{i}|)$. ◇

Below, we will show that, given an ExpTime-admissible tableau system S that is sound and p-complete for some property $\mathcal{P}$ and an input $\mathfrak{i}$ for S, we can construct a tree automaton of size exponential in the size of $\mathfrak{i}$ that accepts exactly the saturated and clash-free S-trees compatible with $\mathfrak{i}$. Together with the emptiness test described in Section 4.1, this shows that the existence of saturated and clash-free S-trees compatible with $\mathfrak{i}$ can be decided in exponential time. Since S is sound and p-complete for $\mathcal{P}$, we have $\mathfrak{i} \in \mathcal{P}$ iff there is a saturated and clash-free S-tree *for* $\mathfrak{i}$. Thus, we must investigate the connection between S-trees for $\mathfrak{i}$ and S-trees compatible with $\mathfrak{i}$. This is done in the next lemma.

Lemma 7.13. There exists a clash-free and saturated S-tree that is *compatible with* $\mathfrak{i}$ iff there exists a clash-free and saturated S-tree *for* $\mathfrak{i}$.

Proof. The "if" direction is straightforward: let $T = ((V, E, n, \ell), \mu)$ be a clash-free and saturated S-tree for $\mathfrak{i}$. Since S is sound and p-complete for $\mathcal{P}$, we can assume w.l.o.g. that the out-degree of the tree of T is bounded by $p(|\mathfrak{i}|)$. It is not hard to show that T is compatible with $\mathfrak{i}$, i. e. satisfies Conditions 1 to 5 of Definition 7.12:

- Each initial S-tree satisfies Conditions 1, 2, and 3 of compatibility, and Condition 3 of admissibility ensures that rule application adds only global memory elements from $\mathit{gme}_S(\mathfrak{i})$, node label elements from $\mathit{nle}_S(\mathfrak{i})$, and edge labels from $\mathit{el}_S(\mathfrak{i})$.
- Each initial S-tree satisfies Condition 4 of compatibility, and rule application cannot delete elements from node labels or from the global memory component.
- Since we assume the out-degree of T to be bounded by $p(|\mathfrak{i}|)$, Condition 5 of compatibility is also satisfied.

For the "only if" direction, let $T = (t, \mu)$ be a clash-free and saturated S-tree that is compatible with $\mathfrak{i}$, and let v_0 be the root of the tree $t = (V, E, n, \ell)$. To construct a clash-free and saturated S-tree for $\mathfrak{i}$, we first construct a (possibly infinite) sequence

$$T_1 \preceq T_2 \preceq T_3 \preceq \cdots$$

of S-trees for $\mathfrak{i}$ such that $T_i \precsim_{\pi_i} T$ for all $i \geq 1$. The construction will be such that the injections π_i that yield $T_i \precsim T$ also build an increasing chain, i. e. π_{i+1} extends π_i for all $i \geq 1$. In the construction, we use a countably infinite set V' from which the nodes of the S-trees T_i are taken. We fix an arbitrary enumeration $x_0, x_1, \ldots$ of V', and write $x < y$ if $x \in V'$ occurs before $y \in V'$ in this enumeration. We then proceed as follows:

- Since T is compatible with $\mathfrak{i}$, there exists $(\Lambda, \nu) \in \mathit{ini}_S(\mathfrak{i})$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$. Define T_1 to be the initial S-tree $((\{x_0\}, \emptyset, \{x_0 \mapsto \Lambda\}, \emptyset), \nu)$. Obviously, $T_1 \precsim_{\pi_1} T$ for $\pi_1 := \{x_0 \mapsto v_0\}$.

- Now, assume that $T_i \precsim_{\pi_i} T$ is already constructed. If T_i is saturated, then T_i is the last S-tree in the sequence. Otherwise, choose the least node x in the tree of T_i (w. r. t. the fixed ordering $<$ on V') such that $P \sim T_i|_x$ for some pattern P that is not saturated, i. e. there exists a rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$. Since $T_i \precsim_{\pi_i} T$, we have $P \precsim T|_{\pi_i(x)}$. Since T is saturated, the pattern $T|_{\pi_i(x)}$ is saturated. By Condition 2 of admissibility, we have $P_j \precsim T|_{\pi_i(x)}$ for some j with $1 \leq j \leq m$. We apply the rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ to x in T_i such that $P_j \sim T_{i+1}|_x$. If the tree of T_{i+1} contains new nodes, then they are taken without loss of generality from V'. Admissibility yields $T_i \preceq T_{i+1}$ and the fact that $P_j \precsim T|_{\pi_i(x)}$ implies that we can define an injection π_{i+1} extending π_i such that $T_{i+1} \precsim_{\pi_{i+1}} T$.

In the definition of the clash-free and saturated S-tree T^* for $\mathfrak{i}$, we distinguish two cases:

1. if the constructed sequence is finite and T_n is the last S-tree in the sequence, then set $T^* := T_n$;

2. otherwise, let T^* be the S-tree T^ω obtained from the sequence $T_1, T_2, \ldots$ as in Case 3 of Definition 7.7.

In both cases, T^* is obviously an S-tree for $\mathfrak{i}$ by definition. In addition, we have $T^* \precsim_\pi T$ where π is the injection obtained as the union of the injections π_i for $i \geq 1$.

It remains to show that T^* is clash-free and saturated. We concentrate on the second case, where $T^* = T^\omega$, since the first case is similar, but simpler. Clash-freeness is an easy consequence of $T^* \precsim T$. In fact, by Condition 4 of admissibility, clash-freeness of T implies that $T^* \precsim T$ is also clash-free.

In order to show saturatedness of T^*, we must look at T^* and its relationship to the S-trees T_i in more detail. Since $T_i \preceq T^* \precsim T$ and the out-degree of the tree of T is bounded by $p(|\mathfrak{i}|)$, the out-degrees of the trees of T_i and T^* are also bounded by $p(|\mathfrak{i}|)$. For a given node x of the tree of T^*, we consider its d-neighbourhood $T^*|_x$.

Since the rules of S only add nodes or elements of node labels or of the global memory component (see Condition 1 in the definition of admissibility), and since the out-degree of x is bounded by $p(|\mathfrak{i}|)$ and the sets $\mathit{nle}_S(\mathfrak{i})$ and $\mathit{gme}_S(\mathfrak{i})$ are finite, there is an i such that x is a node of T_i and "the neighbourhood of x does not change after step i", i. e. $T_i|_x = T_{i+1}|_x = \ldots = T^*|_x$.

Now assume that T^* is not saturated, i. e. there exists a node x in the tree of T^* to which a rule applies, i. e. $P \sim T^*|_x$ for some pattern P with $\mathcal{R}(P) \neq \emptyset$. Let i be such that $T_i|_x = T_{i+1}|_x = \ldots = T^*|_x$. Thus, for $j \geq i$, a rule applies to the node x in the tree of T_i. In the construction of the sequence $T_1, T_2, T_3, \ldots$, we apply a rule only to the least node to which a rule is applicable. Consequently, from the i-th step on, we only apply rules to nodes $y \leq x$. Since there are only finitely many such nodes (see the definition of the order $<$ above), there is one node $y \leq x$ to which rules are applied infinitely often. However, each rule application strictly increases the global memory component, the number of nodes in the d-neighbourhood of y, or the label of a node in this d-neighbourhood. This contradicts the fact that the out-degree of the trees of the T_i is bounded by $p(|\mathfrak{i}|)$, all node labels are subsets of the finite set $\mathit{nle}_S(\mathfrak{i})$, and all global memory components are subsets of the finite set $\mathit{gme}_S(\mathfrak{i})$. □

7.2.2 Accepting Compatible S-trees Using Automata

Recall that we assume our tableau system S to be sound and p-complete w. r. t. a property $\mathcal{P}$. By Lemma 7.13, to check whether an input has property $\mathcal{P}$, it thus suffices to verify the existence of a saturated and clash-free S-tree that is compatible with $\mathfrak{i}$. In this section, we show how this can be done using an automata-based approach.

In contrast to patterns, whose trees can have depth up to d, transitions of tree automata (see Definition 4.2) consider only subtrees of depth 1. This property makes it hard to give a direct translation of an input into an automaton that accepts the saturated and clash-free S-trees that are compatible with this input. For this reason, we first introduce a new type of tree automata "with transitions of depth d", and show that they can be translated into "ordinary" tree automata with transitions as in Definition 4.2.

For a set Q and integers d, k, we denote the set of all (full) k-ary trees of depth d with node labels in Q by $\mathsf{T}_k^d(Q)$. If r is an infinite k-ary Q-tree and x a node in r, then $r|_x$ denotes the d-neighbourhood of x, i. e. the full k-ary subtree of t of depth d with root x.

Definition 7.14 (Tree automata with transitions of depth d). A *tree automaton $\mathcal{A} = (Q, \Sigma, I, \Delta)$ with transitions of depth d* working on k-ary Σ-trees consists of a finite set Q of states, a finite alphabet Σ, a set $I \subseteq \mathsf{T}_k^d(Q)$ of initial trees, and a set of transitions $\Delta \subseteq \Sigma \times \mathsf{T}_k^d(Q)$.

A *run* of $\mathcal{A}$ on an Σ-tree t is a mapping $r : K^* \to Q$ (i. e. a k-ary Q-tree) such that $(t(v), r|_v) \in \Delta$ holds for each node v in K^*. It is *successful* if $r|_\varepsilon \in I$. The *language* of k-ary Σ-trees *accepted by* $\mathcal{A}$ is

$$L(\mathcal{A}) := \{t \mid \text{there is a successful run of } \mathcal{A} \text{ on the } k\text{-ary } \Sigma\text{-tree } t\}. \qquad \diamond$$

It is easy to see that *normal* tree automata (as introduced in Definition 4.2) constitute the special case where the transitions are of depth 1. The following lemma shows that tree automata of depth $d > 1$ are *not* more powerful than normal tree automata. We define the *size* of a tree automaton $\mathcal{A} = (Q, \Sigma, I, \Delta)$ as $|\mathcal{A}| := |Q| + |\Sigma| + |I| + |\Delta|$.

Lemma 7.15. Any tree automaton $\mathcal{A}$ of depth $d > 1$ working on k-ary Σ-trees can be reduced in time polynomial in $|\mathcal{A}|^k$ to a normal tree automaton that accepts the same language.

Proof. Let $\mathcal{A} = (Q, \Sigma, I, \Delta)$ be a tree automaton with transitions of depth d. The normal tree automaton $\mathcal{B} = (P, \Sigma, J, \Theta)$ is defined as follows:

- $P := \{t \mid (\sigma, t) \in \Delta \text{ for some } \sigma \in \Sigma\}$;
- $J := I \cap P$;
- $(t_0, \sigma, t_1, \ldots, t_k) \in \Theta$ iff $(\sigma, t_0) \in \Delta$ and $t_1, \ldots, t_k \in P$ are such that t_i coincides with the subtree of t_0 at node i up to depth $d - 1$.

Clearly, $|P|$ is bounded by $|\Delta|$, $|J|$ is bounded by $|I|$, and $|\Theta|$ is bounded by $|P|^k \cdot |\Delta|$. It is also easy to see that $\mathcal{B}$ can be computed in time polynomial in $|\mathcal{A}|^k$.

It remains to show that $L(\mathcal{A})$ is equal to $L(\mathcal{B})$. First, assume that $t \in L(\mathcal{A})$ and that r is a successful run of $\mathcal{A}$ on t. It is easy to see that the following is a successful run of $\mathcal{B}$ on t:

$$S : K^* \to P : v \mapsto r|_v.$$

Second, assume that r' is a successful run of $\mathcal{B}$ on t. If p is an element of $P \subseteq \mathsf{T}_k^d(Q)$, then we denote the label of its root by $\mathsf{rl}(p)$. We claim that the following is a successful run of $\mathcal{A}$ on t:

$$S' : K^* \to Q : v \mapsto \mathsf{rl}(r'(v)).$$

This is an easy consequence of the fact that $S'|_v = r'(v)$ holds for all $v \in K$. □

The next obstacle on our way towards translating an input into an automaton that accepts the saturated and clash-free S-trees that are compatible with this input is that the S-trees introduced in Section 7.1 are not of a fixed arity k and that their edges are labelled, but not ordered.

It is, however, not hard to convert S-trees compatible with a given input into k-ary Σ-trees for appropriate k and Σ. This is achieved by (i) "padding" with additional dummy nodes as in Definition 4.4, and (ii) representing edge labels via node labels.

Definition 7.16 (Padding). Let $\mathfrak{i} \in \mathfrak{I}$ be an input and $t = (V, E, n, \ell)$ be the tree component of an S-tree compatible with $\mathfrak{i}$. Let v_0 denote the root of t. For each $x \in V$, we use $k(x)$ to denote the out-degree of x in t. We assume that the successors of each node $x \in V$ are linearly ordered and that, for each node $x \in V \setminus \{v_0\}$, $s(x) = i$ iff x is the i-th successor of its predecessor. We inductively define a function m from $\{1, \ldots, p(|\mathfrak{i}|)\}^*$ to $V \cup \{\#\}$ (with $\# \notin V$) as follows:[4]

[4]Note that the p used here stems from p-completeness.

- $m(\varepsilon) = v_0$;
- if $m(v) = x \in V$, $(x, y) \in E$, and $s(y) = i$, then $m(vi) = y$;
- if $m(v) = x \in V$ and $k(x) < i$, then $m(vi) = \#$;
- if $m(v) = \#$, then $m(vi) = \#$ for all $i \in \{1, \ldots, p(|\mathfrak{i}|)\}$.

Let $\mathit{tl}_S(\mathfrak{i})$ denote the set $(\wp(\mathit{nle}_S(\mathfrak{i})) \times \mathit{el}_S(\mathfrak{i})) \cup \{(\#, \#)\}$. The *padding* Π_t of t is the $p(|\mathfrak{i}|)$-ary $\mathit{tl}_S(\mathfrak{i})$-tree defined by setting

1. $\Pi_t(\varepsilon) = (n(v_0), e_0)$ where e_0 is an arbitrary (but fixed) element of $\mathit{el}_S(\mathfrak{i})$;
2. $\Pi_t(v) = (n(x), \Theta)$ if $v \neq \varepsilon$, $m(v) = x \neq \#$, and $\ell(y, x) = \Theta$ where y is the predecessor of x in t;
3. $\Pi_t(v) = (\#, \#)$ if $m(v) = \#$.

Given the tree component t of a pattern for $\mathfrak{i}$ of out-degree at most $p(|\mathfrak{i}|)$, its *d-padding* Π_t^d is the full $p(|\mathfrak{i}|)$-ary $\mathit{tl}_S(\mathfrak{i})$-tree *of depth d* obtained by adding the missing nodes with label $(\#, \#)$ and by representing edge labels via node labels, analogous to the definition of Π_t above. ◇

The final obstacle on our way towards translating an input into an automaton that accepts the saturated and clash-free S-trees that are compatible with this input is the presence of the global memory component in our framework. Transitions of automata (even if they are of depth d) are *local*, whereas the notion of saturatedness involves the *global* memory component. For this reason, we define for each input $\mathfrak{i} \in \mathfrak{I}$ and each $\mu \subseteq \mathit{gme}_S(\mathfrak{i})$ an automaton $\mathcal{A}_\mathfrak{i}^\mu$ that accepts a non-empty language iff there exists a saturated and clash-free S-tree that is compatible with $\mathfrak{i}$ and has global memory component μ.

Definition 7.17 (Automaton for input $\mathfrak{i}$ and global memory component μ). Let $\mathfrak{i} \in \mathfrak{I}$ be an input, $h = p(|\mathfrak{i}|)$, and $\mu \subseteq \mathit{gme}_S(\mathfrak{i})$. The automaton $\mathcal{A}_\mathfrak{i}^\mu = (Q, \Sigma, I, \Delta)$ with transitions of depth d is defined as follows:

- $Q := \Sigma := \mathit{tl}_S(\mathfrak{i})$;
- I consists of all elements t of $\mathsf{T}_k^h(Q)$ whose root label is of the form $\mathit{rl}(t) = (\Psi, e_0)$ where Ψ is such that there exists a tuple $(\Lambda, \nu) \in \mathit{ini}_S(\mathfrak{i})$ with $\Lambda \subseteq \Psi$ and $\nu \subseteq \mu$.
- $(\sigma, t) \in \Delta$ iff the following two conditions are satisfied:
  1. $\sigma = \mathit{rl}(t)$;
  2. either all nodes of t are labelled with $(\#, \#)$ or there is a pattern $P^* = (s, \mu)$ that satisfies the following conditions:
     (a) $t = \Pi_s^k$;
     (b) for each pattern P with $P \sim P^*$, P is saturated (i. e. $\mathcal{R}(P) = \emptyset$);
     (c) for each pattern $P \in \mathcal{C}$, we have $P \not\sim P^*$. ◇

The following lemma shows that the automaton $\mathcal{A}_{\mathfrak{i}}^{\mu}$ accepts exactly the paddings of saturated and clash-free S-trees that are compatible with $\mathfrak{i}$ and have global memory component μ. Consequently, it accepts a non-empty set of trees iff there exists a saturated and clash-free S-tree that is compatible with $\mathfrak{i}$ and has global memory component μ.

Lemma 7.18. Let $\mathfrak{i} \in \mathfrak{I}$ be an input and $\mu \subseteq \mathit{gme}_S(\mathfrak{i})$. Then

$$L(\mathcal{A}_{\mathfrak{i}}^{\mu}) = \{\Pi_t \mid (t, \mu) \text{ is a saturated and clash-free } S\text{-tree compatible with } \mathfrak{i}\}.$$

Proof. Firstly, assume that (t, μ) is a saturated and clash-free S-tree compatible with $\mathfrak{i}$. We claim that Π_t itself is a successful run of $\mathcal{A}_{\mathfrak{i}}^{\mu}$ on Π_t. In fact, $\Pi_t|_{\varepsilon} \in I$ is an immediate consequence of Definition 7.16 (padding) and Condition 4 in Definition 7.12 (S-tree compatible with $\mathfrak{i}$). Now, consider some node v of Π_t. The first condition in the definition of Δ is satisfied since we have Π_t as successful run on itself. Thus, consider the second condition. If $\Pi_t(v) = (\#, \#)$, then the definition of padding implies that all the nodes below v also have label $(\#, \#)$, and thus the second condition in the definition of Δ is satisfied. Otherwise, it is easy to see that the pattern P^* defined by Condition 2(a) of the definition of Δ is a d-neighbourhood in t. Since t is saturated and clash-free, P^* thus satisfies (b) and (c) as well. This completes the proof that Π_t is a successful run of $\mathcal{A}_{\mathfrak{i}}^{\mu}$ on Π_t, and thus shows that $\Pi_t \in L(\mathcal{A}_{\mathfrak{i}}^{\mu})$.

Secondly, assume that $\widehat{t}$ is a tree accepted by $\mathcal{A}_{\mathfrak{i}}^{\mu}$. Because of the first condition in the definition of Δ, $\widehat{t}$ itself is a successful run of $\mathcal{A}_{\mathfrak{i}}^{\mu}$ on $\widehat{t}$. The definitions of Q, I, and Δ imply that there is an S-tree $T = (t, \mu)$ compatible with $\mathfrak{i}$ such that $\Pi_t = \widehat{t}$. The tree t can be obtained from $\widehat{t}$ by "reversing" the padding procedure, i. e. removing dummy nodes and labelling the edges. It remains to show that (t, μ) is saturated and clash-free. Thus, consider a node x of t, and let v be the corresponding node in $\Pi_t = \widehat{t}$. Since x is a node in t, the node v has a label different from $(\#, \#)$. It is easy to see that the pattern P^* defined by (a) in the second condition in the definition of the transition relation coincides with $T|_x$. Thus (b) and (c) in this condition imply that no completion rule and no clash-trigger is applicable to x. □

We are now ready to prove the main result of this section: the ExpTime upper bound induced by ExpTime-admissible tableau systems.

Theorem 7.19. Let $\mathfrak{I}$ be a set of inputs, $\mathcal{P} \subseteq \mathfrak{I}$ a property, and p a polynomial. If there exists an ExpTime-admissible tableau system S for $\mathfrak{I}$ that is sound and p-complete for $\mathcal{P}$, then $\mathcal{P}$ is decidable in ExpTime.

Proof. Let $\mathfrak{i} \in \mathfrak{I}$ be an input. To decide whether $\mathfrak{i} \in \mathcal{P}$, we construct for each $\mu \subseteq \mathit{gme}_S(\mathfrak{i})$ the automaton $\mathcal{A}_{\mathfrak{i}}^{\mu}$. By Lemmas 7.13 and 7.18, $\mathfrak{i} \in \mathcal{P}$ iff at least one of these automata accepts a non-empty language.

It remains to show that this algorithm can be executed in exponential time. Let $n = |\mathfrak{i}|$ and $k = p(|\mathfrak{i}|)$. In order to see that each automaton $\mathcal{A}_{\mathfrak{i}}^{\mu}$ can be constructed in time exponential in n, note that, by Conditions 2 and 3 of ExpTime-admissibility, we can compute $\wp(\mathit{nle}_S(\mathfrak{i}))$ and $\mathit{el}_S(\mathfrak{i})$ in time exponential in n, and thus the same

holds for $\mathit{tl}_S(\mathfrak{i}) = Q = \Sigma$. By Condition 2, to show that I can be computed in exponential time it suffices to show that $\mathsf{T}^h_k(Q)$ is of size exponential in n. This is the case since $|\mathsf{T}^h_k(Q)| = |Q|^{k^{h+1}-1}$, $|Q|$ is exponential in n, k is polynomial in n, and h is a constant. The transition relation Δ can be computed in exponential time due to Conditions 5 and 6 of ExpTime-admissibility and the facts that $|\Delta| \leq |\Sigma| \cdot |\mathsf{T}^h_k(Q)|$, p is a polynomial and h is a constant. Since the automaton $\mathcal{A}^\mu_\mathfrak{i}$ can be computed in exponential time, its size is at most exponential in $|\mathfrak{i}|$. Thus, Lemma 7.15 and the fact that the emptiness test for tree automata can be realized in polynomial time (Vardi and Wolper, 1986) imply that emptiness of each automaton $\mathcal{A}^\mu_\mathfrak{i}$ can be tested in time exponential in the size of the input. By Condition 4 of ExpTime-admissibility we can enumerate all global memory components $\mu \subseteq \mathit{gme}_S(\mathfrak{i})$ in exponential time, and there are exponentially many of them. Thus, the algorithm performs exponentially many ExpTime tests, which is still in ExpTime. □

Since we have shown that the tableau system $S_{\mathcal{ALC}}$ is ExpTime-admissible as well as sound and p-complete (for some polynomial p) for satisfiability of $\mathcal{ALC}$-concepts, we can immediately put Theorem 7.19 to work:

Corollary 7.20. $\mathcal{ALC}$-concept satisfiability w. r. t. general TBoxes is in ExpTime.

This concludes the section dealing with determining the complexity class of algorithms formalised as tableau systems. In the next section, our aim is to derive a practically useable decision procedure from a tableau system.

7.3 Tableau Algorithms from Tableau Systems

The tableau systems introduced in Section 7.1 cannot immediately be used as tableau algorithms since rule application need not terminate. The purpose of this section is to show that, under certain natural conditions, the addition of a straightforward cycle detection mechanism, similar to blocking (Definition 3.5), turns tableau systems into (terminating) decision procedures. In contrast to the ExpTime algorithm constructed in the previous section, the procedures obtained here are usually not worst-case optimal, due to the difficulties of proving ExpTime upper bounds with tableau algorithms described in Section 3.2.2.

Fix a set of inputs $\mathfrak{I}$ and a tableau system $S = (\mathit{NLE}, \mathit{GME}, \mathit{EL}, d, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$. As in the previous section, we require that S has a number of computational properties. Since we do consider decidability rather than complexity issues in this section, it is sufficient for our purposes to impose effectiveness (and not efficiency) constraints. We start with modifying Definition 7.11 (ExpTime-admissible tableau system):

Definition 7.21 (Recursive tableau system). S is called *recursive* iff the following conditions are satisfied:

1. S is admissible (see Definition 7.4);
2. $\mathit{ini}_S(\mathfrak{i})$ can be computed effectively;

3. for each pattern P it can be checked effectively whether, for all patterns P', $P' \sim P$ implies $\mathcal{R}(P') = \emptyset$; if this is not the case, then we can effectively determine a rule
$$P' \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$$
and a bijection π such that $P' \sim_\pi P$.

4. for each pattern P it can be checked effectively whether there is a clash-trigger $P' \in \mathcal{C}$ such that $P' \sim P$. ◇

The main difference between this definition and Definition 7.11 is Condition 3, which now requires that, besides checking the applicability of rules, we can effectively apply at least one rule whenever some rule is applicable at all. Another difference is that we do not actually need to compute the sets $\mathsf{el}_S(\mathfrak{i})$, $\mathsf{nle}_S(\mathfrak{i})$, and $\mathsf{gme}_S(\mathfrak{i})$ in order to apply rules.

Analogously to the case of ExpTime-admissibility, it can be verified that the tableau system $S_{\mathcal{ALC}}$ is recursive. In particular, for the second part of Condition 3 we can again use the fact that the rules of $S_{\mathcal{ALC}}$ are invariant under isomorphism of patterns: this means that it suffices to compute, for a given non-saturated pattern P, a set of patterns $\{P_1, \ldots, P_m\}$ such that $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$. It is easy to see that this can be effectively done for the rules of $S_{\mathcal{ALC}}$.

We now define a more relaxed variant of Definition 7.10 (p-complete).

Definition 7.22 (f-complete). Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function. The tableau system S is called *f-complete for $\mathcal{P}$* iff, for any $\mathfrak{i} \in \mathcal{P}$, there exists a saturated and clash-free S-tree for $\mathfrak{i}$ with out-degree bounded by $f(|\mathfrak{i}|)$. ◇

Since we have already shown that $S_{\mathcal{ALC}}$ is p-complete for some polynomial p, $S_{\mathcal{ALC}}$ is clearly f-complete for the (computable) function f induced by the polynomial p.

In order to implement a cycle detection mechanism, we introduce the notion of blocking: given an S-tree $T = (t, \mu)$, where $t = (V, E, n, \ell)$, we denote by E^* the transitive and reflexive closure of E and say that $x \in V$ is *blocked* iff there exist $u, v \in V$ such that

- uE^*x and vE^*x;
- uE^*v and the path from u to v is of length $\geq d$; and
- $(T|_u)_{d-1} \sim (T|_v)_{d-1}$, where $(T|_u)_{d-1}$ and $(T|_v)_{d-1}$ denote the $d-1$ neighbourhoods of u and v in T, respectively.

Note that, for $d = 1$, this blocking condition reduces to $u \neq v$ and $n(u) = n(v)$, which corresponds to the well-known "equality-blocking" technique that is employed in Section 3.2.2 and appears in various DL tableau algorithms (see e. g. Horrocks and Sattler, 1999; Baader and Sattler, 2001). For $d = 2$, we obtain a more general variant of the "double-blocking" mechanism used for description logics such as $\mathcal{SHIQ}$ (Horrocks et al., 1999). Our version is more general since, in the double-blocking

Preconditions: Let $\mathfrak{I}$ be a set of inputs, $\mathcal{P} \subseteq \mathfrak{I}$ a property, f a recursive function, and S a recursive tableau system for $\mathfrak{I}$ that is sound and f-complete for $\mathcal{P}$.

Algorithm: Return *true* on input $\mathfrak{i} \in \mathfrak{I}$ if the procedure *tableau*(T) defined below returns *true* for at least one initial S-tree T for $\mathfrak{i}$. Otherwise return *false*.

procedure *tableau*(T)

If $P \sim T|_x$ for some $P \in \mathcal{C}$ and node x in T or the out-degree of T exceeds $f(|\mathfrak{i}|)$, then return *false*.

If no rule is applicable to a non-blocked node x in T, then return *true*.

Take a a non-blocked node x in T and a rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ with $P \sim T|_x$.

Let T_i be the result of applying the above rule such that $P_i \sim T_i|_x$, for $1 \leq i \leq m$.

If at least one of *tableau*(T_1), *tableau*(T_2), $\ldots$, *tableau*(T_m) returns *true*, then return *true*.

Return *false*.

Figure 7.2: Decision procedure for $\mathcal{P}$

variant, the isomorphic 2-neighbourhoods in the third item above would be smaller and contain only a single node on depth 1.

The tableau algorithm for $\mathcal{P}$ induced by the tableau system S is described in Figure 7.2. Note that the selection of rules and nodes in the procedure *tableau* is *don't-care*-non-deterministic: for the soundness and completeness of the algorithm, it does not matter which rule we apply when to which node.

Let us verify that the individual steps performed by the algorithm in Figure 7.2 are actually effective:

- the initial trees for an input $\mathfrak{i}$ can be computed effectively, since $\mathit{ini}_S(\mathfrak{i})$ can be computed effectively by Condition 2 of Definition 7.21;
- the condition in the first "if" statement can be checked effectively by Condition 4 of Definition 7.21 and since f is a recursive function;
- the applicability of rules can be checked by the first part of Condition 3 of Definition 7.21;
- finally, that we can effectively take a rule and apply it to a node x follows from the second part of Condition 3 of Definition 7.21.

We now turn to termination, soundness, and completeness of the algorithm.

Lemma 7.23 (Termination). Suppose the preconditions of Figure 7.2 are satisfied. Then the algorithm of Figure 7.2 terminates for any input $\mathfrak{i} \in \mathfrak{I}$.

Proof. Let $\mathfrak{i} \in \mathfrak{I}$. The number of initial trees for $\mathfrak{i}$ is finite and can be computed effectively. Hence, it is sufficient to show that the procedure *tableau* terminates on any initial tree for $\mathfrak{i}$. For each step in which the procedure does not immediately return *true* or *false*, nodes are added to the tree, $n(x)$ properly increases for some nodes x, or μ properly increases (due to Condition 1 of admissibility). Since $n(x) \subseteq \wp(\mathit{nle}_S(\mathfrak{i}))$ holds for any node x and $\mu \subseteq \wp(\mathit{gme}_S(\mathfrak{i}))$ holds for any tree constructed during a run of *tableau*, it is sufficient to show that both the out-degree and the depth of the trees constructed are bounded. The out-degree of the trees is bounded by $f(|\mathfrak{i}|)$ (more precisely, as soon as one rule application yields a tree with out-degree larger than $f(|\mathfrak{i}|)$, the algorithm returns false in the next step). Due to the blocking condition, the length of E-paths does not exceed the number of pairwise non-isomorphic labelled trees (V, E, n, ℓ) of depth up to $d-1$ and out-degree up to $f(|\mathfrak{i}|)$ such that $\mathit{ran}(n) \subseteq \wp(\mathit{nle}_S(\mathfrak{i}))$ and $\mathit{ran}(\ell) \subseteq \mathit{el}_S(\mathfrak{i}))$. □

Lemma 7.24 (Soundness). Suppose the preconditions of Figure 7.2 are satisfied. If the algorithm of Figure 7.2 returns *true* on input $\mathfrak{i}$, then $\mathfrak{i} \in \mathcal{P}$.

Proof. Suppose the algorithm returns *true* on input $\mathfrak{i}$. Then the algorithm terminates with a clash-free S-tree $T = (t, \mu)$, $t = (V, E, n, \ell)$, whose out-degree does not exceed $f(|\mathfrak{i}|)$ and such that no rule is applicable to a non-blocked node in T. As S is sound for $\mathcal{P}$, it is sufficient to show that there exists a saturated and clash-free S-tree for $\mathfrak{i}$. To this end, we construct a clash-free and saturated S-tree

$$T' = ((V', E', n', \ell'), \mu)$$

that is compatible with $\mathfrak{i}$ (from which, by Lemma 7.13, we obtain a clash-free and saturated S-tree for $\mathfrak{i}$). Say that a node $x \in V$ is *directly blocked* if it is blocked but its predecessor is not blocked. For any such x pick a y with yE^*x such that the path from y to x has length $\geq d$ and $(T|_x)_{d-1} \sim (T|_y)_{d-1}$, and say that x is *blocked by* y. Now, we define V' by *unravelling* V, similar to the technique in the proof of Lemma 6.23: V' consists of all non-empty sequences $\langle v_0, x_1, \ldots, x_n \rangle$, where v_0 is the root of V, the nodes $x_1, \ldots, x_n \in V$ are directly blocked or not blocked, and $(x_i, x_{i+1}) \in E$ if x_i is not blocked or x_i is blocked by some $y \in V$ such that $(y, x_{i+1}) \in E$. Define E' by setting, for $\vec{x} = \langle v_0, x_1, \ldots, x_n \rangle \in V'$ and $\vec{y} \in V'$, $(\vec{x}, \vec{y}) \in E'$ iff there exists x_{n+1} such that $\vec{y} = \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle$. Define n' by setting $n'(\langle v_0, x_1, \ldots, x_n \rangle) = n(x_n)$. Finally, define ℓ' by

- $\ell'(\langle v_0, x_1, \ldots, x_n \rangle, \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle) = \ell(x_n, x_{n+1})$ if x_n is not blocked;
- $\ell'(\langle v_0, x_1, \ldots, x_n \rangle, \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle) = \ell(y, x_{n+1})$ if x_n is blocked and y blocks x_n.

We show that T' is a clash-free and saturated S-tree that is compatible with $\mathfrak{i}$. Compatibility is readily checked using the definition of T'. Since T is clash-free and no rule is applicable to a non-blocked node of T, we can prove clash-freeness and saturatedness of T' by showing that any S-pattern P that matches $T'|_{\vec{x}}$ for some node $\vec{x}$ in t' also matches a neighbourhood $T|_x$ for some non-blocked node x in t. Then, one can easily show by induction on m for $0 \leq m \leq d$ and any $\langle v_0, x_0, \ldots, x_n \rangle \in V'$ that

- $(T'|_{\langle v_0,x_1,\ldots,x_n\rangle})_m \sim (T|_{x_n})_m$ if x_n is not blocked and
- $(T'|_{\langle v_0,x_1,\ldots,x_n\rangle})_m \sim (T|_y)_m$ if x_n is blocked by y.

The base case ($m = 0$) is trivial. Thus, consider the induction step from $m-1$ to m for $m - 1 < d$.[5]

Firstly, consider the case where x_n is not blocked. By definition of T', the label of the root node $\langle v_0, x_1, \ldots, x_n\rangle$ of $(T'|_{\langle v_0,x_1,\ldots,x_n\rangle})_m$ coincides with the label of the root node x_n of $(T|_{x_n})_m$. Thus, it is sufficient to show that the respective successor nodes have isomorphic neighbourhoods of depth $m-1$. Let z be a successor of x_n in T, and let $\langle v_0, x_1, \ldots, x_n, z\rangle$ be the corresponding successor of $\langle v_0, x_1, \ldots, x_n\rangle$ in T' (which exists since x_n is not blocked). If z is not blocked, the induction yields $(T'|_{\langle v_0,x_1,\ldots,x_n,z\rangle})_{m-1} \sim (T|_z)_{m-1}$ and we are done. Otherwise, z is blocked by some node y. By induction, we know that $(T'|_{\langle v_0,x_1,\ldots,x_n,z\rangle})_{m-1} \sim (T|_y)_{m-1}$. In addition, the facts that y blocks z and that $m - 1 \leq d - 1$ imply that $(T|_y)_{m-1} \sim (T|_z)_{m-1}$. Thus, we also have $(T'|_{\langle v_0,x_1,\ldots,x_n,z\rangle})_{m-1} \sim (T|_z)_{m-1}$ in this case.

Secondly, consider the case where x_n is blocked by some node y. Let $\langle v_0, x_1, \ldots, x_i, y\rangle$ be the node in T' corresponding to y. By construction of T' we have $(T'|_{\langle v_0,x_1,\ldots,x_i,y\rangle})_m \sim (T'|_{\langle v_0,x_1,\ldots,x_n\rangle})_m$. Thus, it is sufficient to show that $(T'|_{\langle v_0,x_1,\ldots,x_i,y\rangle})_m \sim (T|_y)_m$. Since y is not blocked, this is an instance of the first case in the induction step, which we have already shown.

This finishes the induction proof. It follows that for an S-pattern P, we can deduce from $P \sim T'|_{\langle v_0,x_1,\ldots,x_n\rangle}$ that $P \sim T|_{x_n}$ holds if x_n is not blocked and that $P \sim T|_y$ holds if x_n is blocked by y. □

Lemma 7.25 (Completeness). Suppose the preconditions of Figure 7.2 are satisfied. If $\mathfrak{i} \in \mathcal{P}$, then the algorithm of Figure 7.2 returns *true* on input $\mathfrak{i}$.

Proof. Suppose $\mathfrak{i} \in \mathcal{P}$. Since S is f-complete for $\mathcal{P}$, there exists a clash-free and saturated S-tree $T = (t, \mu)$, $t = (V, E, n, \ell)$, for $\mathfrak{i}$ whose out-degree does not exceed $f(|\mathfrak{i}|)$. We use T to "guide" the algorithm to an S-tree of out-degree at most $f(|\mathfrak{i}|)$ in which no clash-trigger applies and no rule is applicable to a non-blocked node. This will be done in a way such that all constructed S-trees T' satisfy $T' \precsim T$.

For the start, we need to choose an appropriate initial S-tree T_1. Let v_0 be the root of t. Since S-trees for $\mathfrak{i}$ are also compatible with $\mathfrak{i}$, the definition of compatibility implies that there exists $(\Lambda, \nu) \in \mathit{ini}_S(\mathfrak{i})$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$. Define T_1 to be the initial S-tree $((\{v_0\}, \emptyset, \{v_0 \mapsto \Lambda\}, \emptyset), \nu)$. Clearly, $T_1 \precsim T$ holds. We start the procedure *tableau* with the tree T_1.

Now suppose that *tableau* is called with some S-tree T' such that $T' \precsim T$. If no rule is applicable to a non-blocked node in T', we are done: since $T' \precsim T$ and T is clash-free and of out-degree at most $f(|\mathfrak{i}|)$, the same holds for T'. Now suppose that a rule is applicable to a non-blocked node in T'. Assume that the *tableau* procedure has chosen the rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ with $P \sim T'|_x$. Since $T' \precsim_\tau T$ for some τ, we have $P \precsim T|_{\tau(x)}$. Since T is saturated, $T|_{\tau(x)}$ is saturated. By Condition 2 of

[5] Note that the induction step can only be proved for $m - 1 < d$ because the blocking condition ensures isomorphism of neighbourhoods only up to depth $d - 1$.

admissibility, we have $P_j \precsim T|_{\tau(x)}$ for some j, $1 \leq j \leq m$. So we "guide" the *tableau* procedure to continue exploring the S-tree T'_j obtained from T' by applying the rule $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$ such that $P_j \sim T'_j|_x$. Now, $P_j \precsim T|_{\tau(x)}$ implies $T'_j \precsim T$.

Since the *tableau* procedure terminates on any input, the "guidance" process will also terminate and thus succeed in finding an S-tree of out-degree at most $f(|\mathsf{i}|)$ in which no clash-trigger applies and no rule is applicable to a non-blocked node. Hence, *tableau*(T_1) returns *true*. □

The three lemmas just proved imply that we have succeeded in converting the tableau system S into a decision procedure for $\mathcal{P}$.

Theorem 7.26. Suppose the preconditions of Figure 7.2 are satisfied. Then the algorithm of Figure 7.2 effectively decides $\mathcal{P}$.

This concludes the theoretical results of this chapter. In the following, we will illustrate how the tableau systems framework can be put to use by defining tableau systems for two expressive ExpTime-complete DLs.

7.4 Tableau Systems for $\mathcal{SHIO}$ and $\mathcal{SHIQ}$

In this section, we develop the tableau systems $S_{\mathcal{SHIO}}$ and $S_{\mathcal{SHIQ}}$, where the former proves a new ExpTime complexity result for the logic $\mathcal{SHIO}$ and the latter provides an alternative proof of the known ExpTime result for $\mathcal{SHIQ}$ (Tobies, 2001). These two tableau systems exploit different features of the framework: $S_{\mathcal{SHIO}}$ uses the global memory to store and retrieve information about nominals, whereas $S_{\mathcal{SHIQ}}$ uses patterns of size 2 to capture qualifying number restrictions in the presence of inverse roles.

Both $\mathcal{SHIO}$ and $\mathcal{SHIQ}$ are extensions of the DL $\mathcal{SHI}$, which was developed by Horrocks and Sattler (1999) (under the name $\mathcal{ALCHI}_{R^+}$). $\mathcal{SHI}$ extends $\mathcal{ALC}$ with role hierarchies, transitive and inverse roles, and $\mathcal{SHIO}$ additionally allows for nominals, whereas $\mathcal{SHIQ}$ allows for qualifying number restrictions (see Figure 2.2 on Page 29). Since QNR together with role hierarchies and transitive roles lead to undecidability (Horrocks et al., 2000a), it is necessary to prevent an interaction between number restrictions and transitive roles. Therefore, roles that are not transitive and do not have transitive sub-roles are called *simple*, and only simple roles may appear in QNR.

Moreover, QNR can be used to express existential and value restrictions: $\exists \mathsf{r}.\mathsf{C}$ is equivalent to $(\geqslant 1\ \mathsf{r}\ \mathsf{C})$, and $\forall \mathsf{r}.\mathsf{C}$ is equivalent to $(\leqslant 0\ \mathsf{r}\ \neg\mathsf{C})$. Therefore we will define the syntax of $\mathcal{SHIQ}$ in a slightly non-standard way to keep the number of rules small and thus improve the readability of the corresponding tableau system: we do not allow for $\forall$ or $\exists$; instead, we allow for non-simple roles in QNR of the kind $(\leqslant 0\ \mathsf{r}\ \mathsf{C})$ or $(\geqslant 1\ \mathsf{r}\ \mathsf{C})$. Since both $\mathcal{SHIO}$ and $\mathcal{SHIQ}$ allow for internalisation of GCIs as described in Section 2.3.6, we allow only for RBoxes, but not for TBoxes.

Definition 7.27 ($\mathcal{SHIO}$ and $\mathcal{SHIQ}$). Syntax and semantics of $\mathcal{SHIO}$ and $\mathcal{SHIQ}$ are defined as in Figures 2.1 and 2.2, with the modification for existential and value restrictions in $\mathcal{SHIQ}$ mentioned above.

As in Definition 6.15, we refer to the inverse of a role r by $\overline{\mathsf{r}}$. Additionally, for an RBox $\mathcal{B}$, we define the *role hierarchy* $\mathcal{B}^+$ as $\mathcal{B} \cup \{\overline{\mathsf{r}} \sqsubseteq \overline{\mathsf{s}} \mid \mathsf{r} \sqsubseteq \mathsf{s} \in \mathcal{B}\}$, and by $\sqsubseteq^*_{\mathcal{B}}$ we denote the reflexive-transitive closure of $\sqsubseteq$ on $\mathcal{B}^+$. A role r is called *simple* w. r. t. an RBox $\mathcal{B}$ if there exists no role $\mathsf{s} \in \mathsf{N}_{\mathsf{R}^+}$ with $\mathsf{s} \sqsubseteq^*_{\mathcal{B}} \mathsf{r}$ or $\overline{\mathsf{s}} \sqsubseteq^*_{\mathcal{B}} \mathsf{r}$.

In order to capture roles which are implicitly declared to be transitive (e. g. $\overline{\mathsf{r}}$ if $\mathsf{r} \in \mathsf{N}_{\mathsf{R}^+}$), we use, for an RBox $\mathcal{B}$, the predicate $\mathit{trans}_{\mathcal{B}}$, and define that for a role r, $\mathit{trans}_{\mathcal{B}}(\mathsf{r})$ holds iff there exists a role s such that $\mathsf{s} \in \mathsf{N}_{\mathsf{R}^+}$, $\mathsf{s}' \sqsubseteq^*_{\mathcal{B}} \mathsf{r}$ and $\mathsf{r} \sqsubseteq^*_{\mathcal{B}} \mathsf{s}''$ for some $\mathsf{s}', \mathsf{s}'' \in \{\mathsf{s}, \mathsf{s}^-\}$. ◇

As usual, we assume that all concepts are in NNF. Due to the possibility of role inclusion axioms, we cannot restrict the possible node labels to the subconcepts as in Definition 7.2; instead we will define the *closure* clos, which allows for additional value restrictions. These are required by the $\forall_+$-rule (see below), which in turn is necessary to capture transitive sub-roles of non-transitive roles. The global memory is used for three purposes: for transitive roles, role inclusion axioms, and for information about concepts appearing in a node label together with a nominal.

Definition 7.28 (Tableau system $S_{\mathcal{SHIO}}$). For a $\mathcal{SHIO}$ concept C and RBox $\mathcal{B}$, we define the *closure* $\mathit{clos}(\mathsf{C}, \mathcal{B})$ as follows:

- $\mathit{sub}(\mathsf{C}) \subseteq \mathit{clos}(\mathsf{C}, \mathcal{B})$;
- if $\forall \mathsf{r}.\mathsf{D} \in \mathit{clos}(\mathsf{C}, \mathcal{B})$ and the role s appears in C or $\mathcal{B}$, then $\{\forall \mathsf{s}.\mathsf{D}, \forall \overline{\mathsf{s}}.\mathsf{D}\} \subseteq \mathit{clos}(\mathsf{C}, \mathcal{B})$.

We can now define a TS for $\mathcal{SHIO}$, $S_{\mathcal{SHIO}} = (\mathsf{NLE}_{\mathcal{SHIO}}, \mathsf{GME}_{\mathcal{SHIO}}, \mathsf{EL}_{\mathcal{SHIO}}, 1, .^{S_{\mathcal{SHIO}}}, \mathcal{R}_{\mathcal{SHIO}}, \mathcal{C}_{\mathcal{SHIO}})$.

- $\mathsf{NLE}_{\mathcal{SHIO}}$ is the set of all $\mathcal{SHIO}$ concepts,
- $\mathsf{GME}_{\mathcal{SHIO}} = \{(\mathsf{O}, \mathsf{C}) \mid \mathsf{O} \in \mathsf{NOM}$ and $\mathsf{C} \in \mathsf{NLE}_{\mathcal{SHIO}}\} \cup$
  $\{\mathit{trans}(\mathsf{r}) \mid \mathsf{r}$ is a role$\} \cup \{\mathsf{r} \sqsubseteq^* \mathsf{s} \mid \mathsf{r}$ and s are roles$\}$,
- $\mathsf{EL}_{\mathcal{SHIO}}$ is the set of all $\mathcal{SHIO}$ roles, and
- for an input $\mathfrak{i} = (\mathsf{C}, \mathcal{B})$, where C is a concept and $\mathcal{B}$ is an RBox, the function $.^{S_{\mathcal{SHIO}}}$ maps $\mathfrak{i}$ to a tuple $\mathfrak{i}^{S_{\mathcal{SHIO}}} = (\mathit{nle}_{\mathfrak{i}}, \mathit{gme}_{\mathfrak{i}}, \mathit{el}_{\mathfrak{i}}, \mathit{ini}_{\mathfrak{i}})$ with
  - $\mathit{nle}_{\mathfrak{i}} = \mathit{clos}(\mathsf{C}, \mathcal{B})$,
  - $\mathit{el}_{\mathfrak{i}} = \{\mathsf{r} \mid \mathsf{r}$ or $\overline{\mathsf{r}}$ appears in C or $\mathcal{B}\}$,
  - $\mathit{gme}_{\mathfrak{i}} = \{(\mathsf{O}, \mathsf{D}) \mid \mathsf{O} \in \mathsf{NOM} \cap \mathit{clos}(\mathsf{C}, \mathcal{B})$ and $\mathsf{D} \in \mathit{clos}(\mathsf{C}, \mathcal{B})\} \cup$
    $\{\mathit{trans}(\mathsf{r}) \mid \mathsf{r} \in \mathit{el}_{\mathfrak{i}}\} \cup \{\mathsf{r} \sqsubseteq^* \mathsf{s} \mid \{\mathsf{r}, \mathsf{s}\} \subseteq \mathit{el}_{\mathfrak{i}}\}$, and
  - $\mathit{ini}_{\mathfrak{i}} = \{(\{\mathsf{C}\}, \{\mathit{trans}(\mathsf{r}) \mid \mathit{trans}_{\mathcal{B}}(\mathsf{r})$ holds$\} \cup \{\mathsf{r} \sqsubseteq^* \mathsf{s} \mid \mathsf{r} \sqsubseteq^*_{\mathcal{B}} \mathsf{s}$ holds$\})\}$.

R⊓ If $\mathsf{C} \sqcap \mathsf{D} \in n(v)$ for a node $v \in V$ and $\{\mathsf{C}, \mathsf{D}\} \not\subseteq n(v)$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(x) = n(x)$ for all $x \neq v$ and $n'(v) = n(v) \cup \{\mathsf{C}, \mathsf{D}\}$.

R⊔ If $\mathsf{C} \sqcup \mathsf{D} \in n(v)$ and $\{\mathsf{C}, \mathsf{D}\} \cap n(v) = \emptyset$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu), ((V, E, n'', \ell), \mu)\}$,
where $n'(x) = n''(x) = n(x)$ for all $x \neq v$,
$n'(v) = n(v) \cup \{\mathsf{C}\}$ and $n''(v) = n(v) \cup \{\mathsf{D}\}$.

R∃ If $\exists \mathsf{r}.\mathsf{C} \in n(v_0)$, $v_1, \ldots, v_m$ are all the children of v_0 with $\ell(v_0, v_i) = \mathsf{r}$,
and $\mathsf{C} \notin n(v_i)$ for all $i, 1 \leq i \leq m$,
then $\mathcal{R}(P)$ contains the set $\{P_0, P_1, \ldots, P_m\}$ with

- $P_0 = ((V_0, E_0, n_0, \ell_0), \mu)$, where $v' \notin V$, $V_0 = V \cup \{v'\}$,
  $E_0 = E \cup \{(v_0, v')\}, n_0 = n \cup \{v' \mapsto \{\mathsf{C}\}\}, \ell_0 = \ell \cup \{(v_0, v') \mapsto \mathsf{r}\}$.
- for all $i, 1 \leq i \leq m$, $P_i = ((V, E, n_i, \ell), \mu)$,
  where $n_i(x) = n(x)$ for all $x \neq v_i$ and $n_i(v_i) = n(v_i) \cup \{\mathsf{C}\}$.

R∀ If $\forall \mathsf{r}.\mathsf{C} \in n(v)$ for some $v \in V$, v' is an s-neighbour of v with $\mathsf{C} \notin n(v')$ and $\mathsf{s} \sqsubseteq^* \mathsf{r} \in \mu$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(x) = n(x)$ for $x \neq v'$ and $n'(v') = n(v') \cup \{\mathsf{C}\}$.

R∀$_+$ If $\forall \mathsf{r}.\mathsf{C} \in n(v)$, $\{\mathit{trans}(\mathsf{s}), \mathsf{s} \sqsubseteq^* \mathsf{r}, \mathsf{q} \sqsubseteq^* \mathsf{s}\} \subseteq \mu$,
and v' is a q-neighbour of v with $\forall \mathsf{s}.\mathsf{C} \notin n(v')$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(x) = n(x)$ for $x \neq v'$ and $n'(v') = n(v') \cup \{\forall \mathsf{s}.\mathsf{C}\}$.

R↑ If $\{\mathsf{O}, \mathsf{C}\} \subseteq n(v)$ for some $\mathsf{O} \in \mathsf{NOM}$ and $(\mathsf{O}, \mathsf{C}) \notin \mu$,
then $\mathcal{R}(P)$ contains $\{((V, E, n, \ell), \mu')\}$,
where $\mu' = \mu \cup \{(\mathsf{O}, \mathsf{C})\}$.

R↓ If $\mathsf{O} \in n(v)$ for an $\mathsf{O} \in \mathsf{NOM}$, $(\mathsf{O}, \mathsf{C}) \in \mu$ and $\mathsf{C} \notin n(v)$,
then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(x) = n(x)$ for $x \neq v$ and $n'(v) = n(v) \cup \{\mathsf{C}\}$.

Figure 7.3: Tableau rules for $\mathcal{SHIO}$

- The set of clash patterns $\mathcal{C}_{\mathcal{SHIO}}$ contains all patterns $((V, E, n, \ell), \mu)$ of depth 0 with node v_0 such that $\{\mathsf{A}, \neg\mathsf{A}\} \subseteq n(v_0)$ for some concept name A appearing in C.
- The set of rules $\mathcal{R}_{\mathcal{SHIO}}$ is defined in Figure 7.3. For each pattern $P = (t, \mu)$, where $t = (V, E, n, \ell)$ has v_0 as root and depth at most 1, $\mathcal{R}(P)$ contains the described sets. For $\{v, w\} \subseteq V$, we call w an r-*neighbour* of v if $\ell(v, w) = \mathsf{r}$ or $\ell(w, v) = \overline{\mathsf{r}}$. ◇

The rules R⊓, R⊔, R∃ and R∀ are similar to the corresponding rules in $S_{\mathcal{ALC}}$ (see Definition 7.2), with a slight modification to handle role hierarchies properly. The rule R∀$_+$ takes care of transitive roles by propagating value restrictions along edges labelled with transitive roles. Unlike in $S_{\mathcal{ALC}}$, where the global memory is never modified by a rule application, R↑ adds information about nominals to the global memory during the construction of the tableau, which allows us to handle nominals

in a more natural and goal-directed way than by initially guessing the type of each nominal as in Section 5.2.2. From this definition, we obtain our first result:

Lemma 7.29. The TS $S_{\mathcal{SHIO}}$ is admissible, sound and p-complete for $\mathcal{SHIO}$ satisfiability, where $p = (x \mapsto x^2)$.

Proof. Since admissibility of the tableau system is easy to see, we prove only soundness and p-completeness.

Soundness. From a saturated and clash-free S-tree (t, μ) with $t = (V, E, n, \ell)$, we generate a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ as follows: $\Delta^{\mathcal{I}} = \{d_{\mathsf{O}} \mid \mathsf{O} \in \mathsf{NOM}\} \cup \{d_v \mid v \in V \text{ and } n(v) \cap \mathsf{NOM} = \emptyset\}$, i. e. we have one individual for each nominal name and one individual for every tree node that is not labelled with a nominal. A concept name A is interpreted as follows: for every $\mathsf{O} \in \mathsf{NOM} \cap \mathit{clos}(\mathsf{C}, \mathcal{B})$, $d_{\mathsf{O}} \in \mathsf{A}^{\mathcal{I}}$ iff there is a node $v \in V$ with $\{\mathsf{O}, \mathsf{A}\} \subseteq n(v)$. Since $\mathsf{R}\!\uparrow$ and $\mathsf{R}\!\downarrow$ are not applicable, all nodes whose labels contain the same nominal symbol have exactly the same label, and thus $\cdot^{\mathcal{I}}$ is well-defined. For all other individuals, $d_v \in \mathsf{A}^{\mathcal{I}}$ iff $\mathsf{A} \in n(v)$. For a role name r, $\mathsf{r}^{\mathcal{I}}$ is the smallest set satisfying the following conditions: if $\ell(v, w) = \mathsf{r}$ or $\ell(w, v) = \overline{\mathsf{r}}$, then $(d_v, d_w) \in \mathsf{r}^{\mathcal{I}}$; if $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$, then $\mathsf{s}^{\mathcal{I}} \subseteq \mathsf{r}^{\mathcal{I}}$; if $\mathit{trans}(\mathsf{r}) \in \mu$, then $\mathsf{r}^{\mathcal{I}}$ is closed under transitivity.

We will now show by structural induction that complex concepts are interpreted correctly. By definition, all individuals belong to the interpretation of the concept names in their labels, and the interpretation of a nominal contains exactly one element. From our construction, it follows directly that the role hierarchy is respected and that transitive roles are interpreted correctly. For a conjunct $\mathsf{C} \sqcap \mathsf{D}$ (disjunct $\mathsf{C} \sqcup \mathsf{D}$) in a node label $n(v)$, since $\mathsf{R}\!\sqcap$ ($\mathsf{R}\!\sqcup$) is not applicable, it follows that C and D (C or D) are contained in $n(v)$, and by induction, d_v is contained in $\mathsf{C}^{\mathcal{I}} \cap \mathsf{D}^{\mathcal{I}}$ ($\mathsf{C}^{\mathcal{I}} \cup \mathsf{D}^{\mathcal{I}}$).

If $\exists \mathsf{r}.\mathsf{C} \in n(v)$, we assume w.l.o.g. that r is a role name (if it is an inverse role, the argument is analogous). Since $\mathsf{R}\!\exists$ is not applicable, there exists an r-child w of v with $\mathsf{C} \in n(w)$. By construction, it follows that $(d_v, d_w) \in \mathsf{r}^{\mathcal{I}}$ and by induction, $d_w \in \mathsf{C}^{\mathcal{I}}$. If $\forall \mathsf{r}.\mathsf{C} \in n(v)$, we again assume that r is a role name. There are two possible reasons why (d_v, d_w) can be contained in $\mathsf{r}^{\mathcal{I}}$: firstly, if w is an s-neighbour of v for some s with $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$. In this case, it follows that $\mathsf{C} \in n(w)$ because $\mathsf{R}\forall$ is not applicable, and thus $d_w \in \mathsf{C}^{\mathcal{I}}$. Secondly, if there exist roles $\mathsf{s}, \mathsf{s}_1, \ldots, \mathsf{s}_m$ such that $\{\mathit{trans}(\mathsf{s}), \mathsf{s} \sqsubseteq^{*} \mathsf{r}, \mathsf{s}_i \sqsubseteq^{*} \mathsf{s}\} \subseteq \mu$ for all $i \in \{1, \ldots, m\}$ and there is an s_i-chain from v to w, i. e. a sequence of nodes $v_1, v_2, \ldots, v_n$ such that, for all edges $e \in \{(v, v_1), (v_1, v_2), \ldots, (v_n, w)\}$, it holds that $e \in E$ and $\ell(e) = \mathsf{s}_i$ for some i. In this case, since $\mathsf{R}\forall_+$ is not applicable, all nodes $v_1, \ldots, v_m$ are labelled with $\forall \mathsf{s}.\mathsf{C}$ and, since $\mathsf{R}\forall$ is not applicable to v_m, $n(w)$ contains C. By induction, it follows that $d_v \in (\forall \mathsf{r}.\mathsf{C})^{\mathcal{I}}$.

Completeness. We have to show that if there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ for an input $\mathsf{i} = (\mathsf{C}, \mathcal{B})$, then there also exists a clash-free and saturated S-tree (t, μ) with $t = (V, E, n, \ell)$ for i, whose width is at most quadratic in $|\mathsf{i}|$. We will create (t, μ) by unravelling $\mathcal{I}$: firstly, we add the appropriate transitivity axioms ($\mathit{trans}(\mathsf{r})$ if $\mathit{trans}_{\mathcal{B}}(\mathsf{r})$ holds) and role inclusion axioms ($\mathsf{r} \sqsubseteq^{*} \mathsf{s}$ if $\mathsf{r} \sqsubseteq^{*}_{\mathcal{B}} \mathsf{s}$ holds) to μ. The tree t is inductively

defined as follows: since $\mathcal{I}$ is a model for i, there is an individual d_0 in $\Delta^{\mathcal{I}}$ which satisfies C. We start with $V = \{v_0\}$ and define $n(v_0)$ as the set of all concepts in $\mathit{clos}(\mathsf{C}, \mathcal{B})$ which d_0 satisfies. We define a function $\pi : V \to \Delta^{\mathcal{I}}$ and set $\pi(v_0) = d_0$.

Then we iterate, for every node v, the following procedure: for every existential restriction $\exists \mathsf{r}.\mathsf{D} \in n(v)$ we choose a witness individual $d \in \Delta^{\mathcal{I}}$ with $d \in \mathsf{D}^{\mathcal{I}}$ and $(\pi(v), d) \in \mathsf{r}^{\mathcal{I}}$ (such a witness exists by definition of $n(v)$). We create a new node w with $\pi(w) = d$, $(v, w) \in E$ and $\ell(v, w) = \mathsf{r}$. Again, we label w with the appropriate concepts in $\mathit{clos}(\mathsf{C}, \mathcal{B})$ and then continue the iteration. For every nominal concept O, we add to μ the pair (O, D) for every concept $\mathsf{D} \in \mathit{clos}(\mathsf{C}, \mathcal{B})$ that the unique element d_{O} of $\mathsf{O}^{\mathcal{I}}$ satisfies.

It is easy to see that (t, μ) is compatible with i and clash-free. We will now show that it is also saturated: from the definition of clos, it follows that $\mathsf{R}\sqcap$ and $\mathsf{R}\sqcup$ are not applicable. If a node label $n(v)$ contains a concept $\exists \mathsf{r}.\mathsf{D}$ then, by construction of t, there is an r-successor of v labelled with D. Likewise, if $\forall \mathsf{r}.\mathsf{D} \in n(v)$ then all r-neighbours of v are labelled with D. If $\forall \mathsf{r}.\mathsf{D} \in n(v)$ then μ contains $\mathsf{s} \sqsubseteq^* \mathsf{r}$, $\mathsf{q} \sqsubseteq^* \mathsf{s}$, and $\mathit{trans}(\mathsf{s})$, and there is a q-neighbour w of v then, since $\mathcal{I}$ is a model, $(\pi(v), \pi(w)) \in \mathsf{s}^{\mathcal{I}}$ and, since $\mathsf{s}^{\mathcal{I}}$ is transitive, it also holds for every node u with $(\pi(w), \pi(u)) \in \mathsf{s}^{\mathcal{I}}$ that $(\pi(v), \pi(u)) \in \mathsf{s}^{\mathcal{I}}$, which implies $\pi(u) \in \mathsf{D}^{\mathcal{I}}$. Thus it follows that $\pi(w) \in (\forall \mathsf{r}.\mathsf{D})^{\mathcal{I}}$ and, since $\mathsf{s} \sqsubseteq^*_{\mathcal{B}} \mathsf{r}$, $v(w)$ contains $\forall \mathsf{s}.\mathsf{D}$, which means that $\mathsf{R}\forall_+$ is not applicable. Finally, since every node n with $\pi(n) = d_{\mathsf{O}}$ for a nominal O is labelled with exactly those concepts for which μ contains (O, C), $\mathsf{R}\uparrow$ and $\mathsf{R}\downarrow$ are not applicable.

The width of the S-tree is quadratic in the length of i because we create for every node at most one successor for every existential restriction in $\mathit{clos}(\mathsf{i})$ and the number of such concepts is bounded by the product of the number of roles appearing in C or $\mathcal{B}$ and the number of existential subconcepts of C. □

From Lemma 7.29, we can derive that $\mathcal{SHIO}$ satisfiability is decidable through a tableau algorithm, and we know that for the blocking condition, equality-blocking suffices, i. e. we do not require double-blocking as e. g. for $\mathcal{SHIQ}$ (Horrocks et al., 2000a), since we use only patterns of depth at most 1. We can also derive a complexity result:

Theorem 7.30. Satisfiability for $\mathcal{SHIO}$ concepts w. r. t. RBoxes is ExpTime-complete.

Proof. It is easy to see that S is ExpTime-admissible: e. g. the size of $\mathit{nle}_{\mathcal{S}}(\mathsf{i})$ and $\mathit{gme}_{\mathcal{S}}(\mathsf{i})$ is quadratic in the size of the input. Soundness and completeness have been shown above. ExpTime-hardness follows from the fact that $\mathcal{SHIO}$ is an extension of $\mathcal{ALC}$ with TBoxes, for which the satisfiability problem is known to be ExpTime-hard (Schild, 1994). □

For $\mathcal{SHIQ}$, we need a TS with quite different properties: in order to handle QNR in the presence of inverse roles correctly, we require patterns of size 2. However, this makes a special treatment for the root node necessary since it does not have a predecessor. We therefore introduce an additional concept name ROOT and a special

R⊓/R⊔ See $\mathcal{R}_{\mathcal{SHIO}}$.

RC If $(\gtrless m\ \mathsf{r}\ \mathsf{C}) \in n(v)$ (where $\gtrless$ is a placeholder for $\geqslant$ or $\leqslant$)
for some m and a node $v \in V$ and $\{\mathsf{C}, \doteq\neg\mathsf{C}\} \cap n(w) = \emptyset$ for an r-neighbour w of v,
then $\mathcal{R}$ contains the set $\{((V, E, n', \ell), \mu),\ ((V, E, n'', \ell), \mu)\}$,
where $n'(x) = n''(x) = n(x)$ for all $x \in V \setminus \{w\}$ and $n'(w) = n(w) \cup \{\mathsf{C}\}$
and $n''(w) = n(w) \cup \{\doteq\neg\mathsf{C}\}$.

R∀$_+$ If $(\leqslant 0\ \mathsf{r}\ \mathsf{C}) \in n(v)$ for a node $v \in V$ and there is a role s
with $\{\mathit{trans}(\mathsf{s}),\ \mathsf{q} \sqsubseteq^{*} \mathsf{s},\ \mathsf{s} \sqsubseteq^{*} \mathsf{r}\} \subseteq \mu$ and a q-neighbour w of v with $(\leqslant 0\ \mathsf{s}\ \mathsf{C}) \notin n(w)$,
then $\mathcal{R}$ contains $\{((V, E, n', \ell), \mu)\}$,
where $n'(x) = n(x)$ for $x \neq w$ and $n'(w) = n(w) \cup \{(\leqslant 0\ \mathsf{s}\ \mathsf{C})\}$.

R$\geqslant$ If P is a pattern of depth 2 and $(\geqslant m\ \mathsf{r}\ \mathsf{C}) \in n(w)$ for a successor w of v_0
and there are less than m s-neighbours of w with $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$ and $\mathsf{C} \in n(u_i)$,
then $\mathcal{R}$ contains the set $\{P_1, \ldots, P_n, P_{n+1}, \ldots, P_{n+h}\}$,
where $u_1 \ldots u_n$ are the s-neighbours of w with $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$ and $\mathsf{C} \notin n(u_i)$;
$\mathsf{s}_1, \ldots, \mathsf{s}_h$ are all roles such that $\mathsf{s}_i \sqsubseteq^{*} \mathsf{r} \in \mu$; and

- For $1 \leq i \leq n$, $P_i = ((V, E, n_i, \ell), \mu)$ with $n_i(x) = n(x)$ for all $x \in V \setminus \{u_i\}$
and $n_i(u_i) = n(u_i) \cup \{\mathsf{C}\}$.
- For $n+1 \leq i \leq n+h$, $P_i = ((V_i, E_i, n_i, \ell_i), \mu)$ with $u_i \notin V$,
$V_i = V \cup \{u_i\}$, $E_i = E \cup \{(w, u_i)\}$, $n_i(x) = n(x)$ for all $x \in V$, $n_i(u_i) = \{\mathsf{C}\}$,
and $\ell_i = \ell \cup \{(w, u_i) \mapsto \mathsf{s}_{i-n}\}$.

R$\geqslant_{\mathsf{ROOT}}$ If $\{(\geqslant m\ \mathsf{r}\ \mathsf{C}), \mathsf{ROOT}\} \in n(v_0)$ of the root node v_0
and there are less than m s-successors of v_0 with $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$ and $\mathsf{C} \in n(u_i)$,
then $\mathcal{R}$ contains the set $\{P_1, \ldots, P_n, P_{n+1}, \ldots, P_{n+h}\}$,
where $u_1 \ldots u_n$ are the s-successors of v_0 with $\mathsf{s} \sqsubseteq^{*} \mathsf{r} \in \mu$ and $\mathsf{C} \notin n(u_i)$;
$\mathsf{s}_1, \ldots, \mathsf{s}_h$ are all sub-roles of r; and $P_1, \ldots, P_{n+h}$ are defined as for R$\geqslant$.

Figure 7.4: Tableau rules for $\mathcal{SHIQ}$

$\geqslant$-rule for the root node. In contrast to the algorithm by Horrocks et al. (2000a), we do not have a $\leqslant$-rule, but a non-deterministic $\geqslant$-rule, which recycles existing neighbour nodes if necessary, similar to the non-deterministic $\exists$-rules in $S_{\mathcal{ALC}}$ and $S_{\mathcal{SHIO}}$. (A rule merging two nodes would violate Condition 1 of admissibility in Definition 7.4.) Moreover, when generating a new successor in order to satisfy a concept $(\geqslant n\ \mathsf{r}\ \mathsf{C})$, this rule non-deterministically chooses a sub-role of r (or r itself) for the edge label. This is necessary e. g. to handle a concept like $(\geqslant 1\ \mathsf{r}\ \mathsf{C}) \sqcap (\geqslant 1\ \mathsf{s}\ \mathsf{D}) \sqcap (\leqslant 1\ \mathsf{r}\ \top)$, with $\mathsf{s} \sqsubseteq \mathsf{r}$ properly: if the $\geqslant$-rule is applied first to $(\geqslant 1\ \mathsf{r}\ \mathsf{C})$ then, if r was the only possible edge label, it would be impossible to process $(\geqslant 1\ \mathsf{s}\ \mathsf{D})$ without causing a clash.

As usual for logics allowing for QNR, we require the choose-rule RC (Baader et al., 1996) in order to detect "hidden" clashes in concepts like $(\leqslant 1\ \mathsf{r}\ \mathsf{C}) \sqcap (\leqslant 1\ \mathsf{r}\ \neg\mathsf{C}) \sqcap (\geqslant 3\ \mathsf{r}\ \top)$: if a node label contains an at-most-restriction involving a node r and a concept C, then RC non-deterministically adds C or $\doteq\mathsf{C}$ to every r-neighbour.

Obviously, we can again transform $\mathcal{SHIQ}$ concepts into NNF using the duality of $\leqslant$ and $\geqslant$. The definition of closure is extended as follows in order to handle QNR: if $(\leqslant m\ \mathsf{r}\ \mathsf{D})$ or $(\geqslant m\ \mathsf{r}\ \mathsf{D}) \in \mathit{clos}(\mathsf{C}, \mathcal{B})$, then $\{\mathsf{D}, \doteq\neg\mathsf{D}\} \subseteq \mathit{clos}(\mathsf{C}, \mathcal{B})$, and if $(\leqslant 0\ \mathsf{r}\ \mathsf{D}) \in \mathit{clos}(\mathsf{C}, \mathcal{B})$ and the role s appears in C or $\mathcal{B}$, then $\{(\leqslant 0\ \mathsf{s}\ \mathsf{D}), (\leqslant 0\ \overline{\mathsf{s}}\ \mathsf{D})\} \subseteq \mathit{clos}(\mathsf{C}, \mathcal{B})$.

Definition 7.31 (Tableau System $S_{\mathcal{SHIQ}}$). The TS $S_{\mathcal{SHIQ}} = (\mathsf{NLE}_{\mathcal{SHIQ}}, \mathsf{GME}_{\mathcal{SHIQ}}, \mathsf{EL}_{\mathcal{SHIQ}}, 2, \cdot^{S_{\mathcal{SHIQ}}}, \mathcal{R}_{\mathcal{SHIQ}}, \mathcal{C}_{\mathcal{SHIQ}})$ is defined like $S_{\mathcal{SHIO}}$, with the following exceptions:

- $\mathsf{NLE}_{\mathcal{SHIQ}}$ contains the additional element ROOT,
- $\mathsf{GME}_{\mathcal{SHIQ}}$ and $\mathit{gme}_{\mathfrak{i}}$ do not contain any "nominal elements" (O, C),
- $\mathit{ini}_{\mathfrak{i}} = \{(\{\mathsf{C}, \mathsf{ROOT}\}, \{\mathit{trans}(\mathsf{r}) \mid \mathit{trans}_{\mathcal{B}}(\mathsf{r}) \text{ holds}\} \cup \{\mathsf{r} \sqsubseteq^{*} \mathsf{s} \mid \mathsf{r} \sqsubseteq^{*}_{\mathcal{B}} \mathsf{s} \text{ holds}\})\}$, and
- the set of rules $\mathcal{R}_{\mathcal{SHIQ}}$ is as given in Figure 7.4,
- the set of clash triggers $\mathcal{C}_{\mathcal{SHIQ}}$ contains all patterns in $\mathcal{C}_{\mathcal{SHIO}}$ and additionally all patterns of depth at most 2 such that $(\leqslant m\ \mathsf{s}\ \mathsf{C}) \in n(v)$ with $v \in V$ and there are at least $m+1$ r-neighbours of v with $\mathsf{r} \sqsubseteq^{*} \mathsf{s} \in \mu$. ◇

Please note that no rule modifies μ and that $\mathsf{R}\geqslant$ applies only to patterns whose depth is exactly 2, whereas the other rules apply to patterns of depth up to 2. Here, we obtain an explanation why double-blocking is necessary for $\mathcal{SHIQ}$: the pattern depth required to correctly handle QNR in the presence of inverse roles is 2 (see Section 7.3). Observe also that we do not need a specific rule for concepts of the form $(\leqslant 0\ \mathsf{s}\ \mathsf{D})$ to propagate $\dot{\neg}\mathsf{D}$ to all the appropriate neighbours (analogously to $\mathsf{R}\forall$), since this task is performed by the rule RC. We also do not have a $\leqslant$-rule, but only a corresponding clash-trigger.

For the proof of p-completeness, we require that the numbers in number restrictions are coded in unary, since otherwise the width of a model can be exponential in the size of the input. We can then obtain alternative proofs for the known results of $\mathcal{SHIQ}$ decidability and complexity:

Lemma 7.32. If unary coding is used in number restrictions, the TS $S_{\mathcal{SHIQ}}$ is admissible, sound and p-complete for $\mathcal{SHIQ}$ satisfiability, where $p = (x \mapsto x^2)$.

Proof. Admissibility is again easy to see. The soundness proof is more immediate than for $S_{\mathcal{SHIO}}$, since a completion tree corresponds directly to a model, and we do not have to "merge" nodes labelled with nominals.

Soundness. From a saturated and clash-free S-tree (t, μ) with $t = (V, E, n, \ell)$, we generate a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ as follows: $\Delta^{\mathcal{I}} = \{d_v \mid v \in V\}$. For any concept name D and any role r we have the same interpretation as in the proof of soundness for $\mathcal{SHIO}$. Thus, for $\sqcap$ and $\sqcup$ concepts, the proof is analogous to the one for Lemma 7.29.

We will now show that the QNR $(\leqslant m\ \mathsf{r}\ \mathsf{D})$ and $(\geqslant m\ \mathsf{r}\ \mathsf{D})$ are also interpreted correctly. Let $(\geqslant m\ \mathsf{r}\ \mathsf{C})$ be a concept in $n(v)$ of a node v. Then, since the S-tree is saturated, there exist at least m nodes $u_1, \ldots, u_m$ where for every $1 \leq i \leq m$, there exists a role s such that u_i is an s-neighbour of v, μ contains $\mathsf{s} \sqsubseteq^{*} \mathsf{r}$, and $n(u_i)$ contains C. By induction, we obtain $d_{u_i} \in \mathsf{C}^{\mathcal{I}}$. From our construction, it follows that $(d_v, d_{u_i}) \in \mathsf{r}^{\mathcal{I}}$ for all i, and thus $\#\{d \mid (d_v, d) \in \mathsf{r}^{\mathcal{I}} \text{ and } d \in \mathsf{C}^{\mathcal{I}}\} \geq m$.

If $(\leqslant m\ \mathsf{r}\ \mathsf{C}) \in n(v)$ for a node v and a simple role r then, since the S-tree is clash-free, there exist at most m s_i-neighbours $u_1 \ldots u_m$ of v for some s_i such that

$\mathsf{s}_i \sqsubseteq^* \mathsf{r} \in \mu$ with $\mathsf{C} \in n(u_i)$ and hence $d_{u_i} \in \mathsf{C}^\mathcal{I}$. All other s_i-neighbours w contain the concept $\dot{\neg}\mathsf{C}$ because the rule RC is not applicable, and by induction their corresponding individuals belong to $(\neg\mathsf{C})^\mathcal{I}$. Since r is simple (and thus also all of r's sub-roles), our construction of $\mathsf{r}^\mathcal{I}$ does not introduce any further $\mathsf{r}^\mathcal{I}$-neighbours. Therefore, it follows that $\#\{d \mid (d_v, d) \in \mathsf{r}^\mathcal{I} \text{ and } d \in \mathsf{C}^\mathcal{I}\} \leq m$ holds.

For a concept $(\leqslant 0\ \mathsf{r}\ \mathsf{C}) \in n(v)$ and a non-simple role r, the proof is similar to the one for $\forall$-concepts in $\mathcal{SHIO}$: for roles $\mathsf{s}, \mathsf{s}_1, \ldots, \mathsf{s}_n$ with $\mathit{trans}_\mathcal{B}(\mathsf{s})$ and $\mathsf{s}_i \sqsubseteq^*_\mathcal{B} \mathsf{s} \sqsubseteq^*_\mathcal{B} \mathsf{r}$, it follows from saturatedness of the S-tree that $(\leqslant 0\ \mathsf{r}\ \mathsf{C}) \in n(w)$ for every node w that is reachable from v via an s_i-chain, and thus all r-neighbours of w are labelled with $\dot{\neg}\mathsf{C}$ since the tree is clash-free and the rule RC is not applicable.

Completeness. As for $S_{\mathcal{SHIO}}$, we will create a clash-free and saturated S-tree (t, μ) with $t = (V, E, n, \ell)$ for a satisfiable input $\mathfrak{i} = (\mathsf{C}, \mathcal{B})$ by unravelling a model $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$. The global memory μ is created as in the case of $\mathcal{SHIO}$ by adding all appropriate transitivity and role inclusion axioms. The tree is inductively defined as follows: we start with an individual $d_0 \in \mathsf{C}^\mathcal{I}$, create a node v_0, and a function π with $\pi(v_0) = d_0$. Moreover, we define, for this and all further nodes v, $n(v)$ as the set of all concepts $\mathsf{D} \in \mathit{clos}(\mathsf{C}, \mathcal{B})$ which the individual $\pi(v)$ satisfies. For the root node, we add the marker concept ROOT to $n(v_0)$.

New nodes are added to the tree if there is a node v and a concept $(\geqslant m\ \mathsf{r}\ \mathsf{D}) \in n(v)$: if there exist only n r-neighbours (or s-neighbours with $\mathsf{s} \sqsubseteq^* \mathsf{r} \in \mu$) of v and $n < m$, we choose appropriate individuals $d_{n+1} \ldots d_m$, i. e. individuals with $d_i \in \mathsf{D}^\mathcal{I}$ and $(\pi(v), d_i) \in \mathsf{r}^\mathcal{I}$ to which none of the existing r-neighbours of v is mapped by π. For these nodes, we create new neighbours $u_{n+1} \ldots u_m$ of v and set $\pi(u_i) = d_i$, $n(v) = \{\mathsf{E} \in \mathit{clos}(\mathsf{C}, \mathcal{B}) \mid d_i \in \mathsf{E}^\mathcal{I}\}$, and $\ell(v, u_i) = \mathsf{s}$ for the smallest (w. r. t. $\mathcal{B}$) role s with $\mathsf{s} \sqsubseteq^*_\mathcal{B} \mathsf{r}$ and $(\pi(v), d_i) \in \mathsf{s}^\mathcal{I}$. Since $\mathcal{I}$ is a model, it always is possible to find appropriate individuals. From this construction, it follows that $\mathsf{R}\geqslant$ and $\mathsf{R}\geqslant_\mathsf{ROOT}$ are not applicable. The rule RC is not applicable by construction, since every node satisfies either D or $\dot{\neg}\mathsf{D}$ for any concept $\mathsf{D} \in \mathit{clos}(\mathsf{C}, \mathcal{B})$, and for $\mathsf{R}\sqcap$, $\mathsf{R}\sqcup$ and $\mathsf{R}\forall_+$, saturatedness follows analogous to the proof for $S_{\mathcal{SHIO}}$. Note that the out-degree of the S-tree is bounded by the number of concepts of the form $(\geqslant m\ \mathsf{r}\ \mathsf{D}) \in \mathit{clos}(\mathsf{C}, \mathcal{B})$ and the highest number m occurring in such a concept.

The resulting S-tree can neither contain a clash-trigger with $\{\mathsf{D}, \dot{\neg}\mathsf{D}\} \subseteq n(v)$ for a node v and a concept D (since $\mathcal{I}$ is a model) nor a clash-trigger with a number restriction of the form $\mathsf{E} = (\leqslant m\ \mathsf{r}\ \mathsf{D}) \in n(v)$ of a node $v \in V$ with $\pi(v) = d$ (because $d \in \mathsf{E}^\mathcal{I}$ and we create at most one r-neighbour of v for every $\mathsf{r}^\mathcal{I}$-neighbour of d).

It is easy to see that (t, μ) is compatible with $\mathfrak{i}$ and the width is at most quadratic in length of $\mathfrak{i}$ since we create only m successors for a concept $(\geqslant m\ \mathsf{r}\ \mathsf{D}) \in \mathit{clos}(\mathsf{C}, \mathcal{B})$, the number m is coded in unary and the number of such concepts is quadratic in the length of $\mathfrak{i}$. □

Theorem 7.33. Satisfiability for $\mathcal{SHIQ}$ concepts w. r. t. RBoxes is ExpTime-complete.

Proof. The tableau system $S_{\mathcal{SHIQ}}$ obviously is ExpTime-admissible, e. g. the size of $\mathsf{nle}_S(\mathfrak{i})$ and of $\mathsf{gme}_S(\mathfrak{i})$ is quadratic in the size of the input. Soundness and p-completeness have been shown above. ExpTime-hardness follows as for $S_{\mathcal{SHIO}}$. □

Comparing these proofs with those from Theorems 3.2 and 3.6, which are for significantly less expressive logics, or with the original proofs for the $\mathcal{SHIQ}$ tableau algorithm by Horrocks et al. (1999), which require an involved unravelling procedure of a completion tree with blocked nodes and the definition of structure (there called tableau) ranging between a completion tree and a model, the benefits of defining a tableau system become clear: an blocking condition suitable for the language $\mathcal{L}$ can be derived directly from the tableau system $S_{\mathcal{L}}$; the proof of soundness does not have to take blocking into consideration; and it is only necessary to show ExpTime-admissibility in order to obtain an ExpTime upper bound.

7.5 Chapter Summary

We have developed *tableau systems*, a framework for tableau algorithms deciding satisfiability (or another property under consideration) for ExpTime logics. From an algorithm defined within this framework, we can derive

- an automata algorithm deciding satisfiability in ExpTime,
- an appropriate blocking condition and a terminating tableau algorithm that is well-suited for implementation and likely to exhibit the good performance in practice that is typical for tableau algorithms.

The properties that are required from tableau systems to achieve this result are essentially the following:

- the node and edge labels and the global memory for a specific input can be computed in time at most exponential in the size of the input;
- the width of the completion tree is bounded polynomially by the size of the input;
- applicability of rules and containment of clashes are decidable in exponential time;
- rule applications must add information, and they must do so in such a way that the completion tree approaches saturation (i. e. a situation in which no rule is applicable anymore).

With this framework, we are able to prove a new ExpTime result for $\mathcal{SHIO}$ and provide an alternative proof of the known complexity result for $\mathcal{SHIQ}$. It turns out that these two logics make use of different features of the tableau framework: to capture nominals, we store information in the global memory component, whereas a higher pattern depth is needed to handle QNR properly.

In order to prove termination of tableau systems in general, the framework stipulates that rules always extend the completion tree—which means that we disallow e. g. a $\leqslant$-rule that involves merging two nodes in order to satisfy an at-most restriction. This is necessary because allowing rules to merge nodes requires additional arguments in the termination proof (see e. g. Horrocks et al., 2000a), whereas our aim is to take the obligation to show termination away from the user of the framework and prove termination in general. In order to achieve a framework that is as universal as possible (and in particular not restricted to logics that are invariant under bisimulation; see the corresponding discussion after Definition 7.2), it is necessary to formulate "generating" rules (like $\mathsf{R}\exists$ and $\mathsf{R}\geqslant$) in a non-deterministic way, i. e. they can either generate new nodes or recycle existing ones. A straightforward implementation of the tableau algorithm resulting from such a tableau system therefore would involve unnecessary non-determinism and thus perform sub-optimal in practice. Consequently, for an implementation these rules should be modified to their standard, deterministic versions.

However, the correctness proofs for tableau systems are significantly simplified by the fact that they do not need to take a blocking condition into consideration, and therefore a large amount of work can be avoided, like finding an appropriate blocking condition and proving soundness in the presence of blocking, i. e. unravelling the blocked nodes in the tableau. This tedious and repetitive work was performed for the framework in general (see Section 7.3). We therefore believe that the simplicity of the proofs justifies the additional overhead resulting from the formalisation of the algorithm within the tableau systems framework.

Chapter 8

Conclusion

In this thesis we have examined the relations between tableau and automata algorithms used for deciding the satisfiability problem in the area of description logics. While tableau algorithms test the satisfiability of an input by trying to construct a model according to a set of rules, automata algorithms translate the input into an automaton accepting all models for the input and subsequently test the language accepted by the automaton for emptiness. Although at first glance, the rules of a tableau algorithm look similar to the transitions of the automaton, TAs and AAs behave quite differently in practice and have complementary advantages and disadvantages, which have been described in detail in Chapter 1 and are recapitulated here.

- Tableau algorithms are well-suited to show PSpace upper bounds, but require considerable efforts in order to run in ExpTime. Proving termination for expressive logics that do not have the finite tree model property requires a cycle detection mechanism, which significantly complicates the correctness proofs. In practice however, TAs have the advantage of being amenable to a considerable number of efficient optimisations. Therefore, implementations of tableau algorithms perform surprisingly well on knowledge bases resulting from real-life applications, even for logics with an "intractable" satisfiability problem.
- Automata algorithms employ a deterministic bottom-up emptiness test of the (finite) automaton in order to decide satisfiability of the input. Thus, they handle non-determinism and infinite structures implicitly, which makes them very elegant from a theoretical point of view. Moreover, this property makes AAs well-suited for proving ExpTime upper bounds, but not for lower complexity classes. Their main disadvantage is that they require exponential time even in the best case because the input is translated into an automaton of exponential size. Consequently, one cannot expect acceptable performance in practice from an unoptimised implementation of an automata algorithm.

In Section 1.1, we formulated three questions regarding the relation between tableaus and automata and the possibility of transferring positive properties between the two approaches. In the following, we will summarise the answers that can be obtained from the work presented in this thesis.

1. What are the precise relations between the data structures used in the different approaches? Can a tableau constructed by a TA serve as an input for the corresponding AA? It was observed by Calvanese et al. (2002) that the similarity between tableau and automata algorithms is particularly strong in the case of alternating two-way automata. In Chapter 5, we use a tableau algorithm to perform the emptiness test of such an automaton. It turns out that a tableau generated by the TA looks almost exactly like a strategy tree, which is used in the automata framework as a witness for non-emptiness of the language accepted by the automaton. More precisely, the unravelling of the tableau is identical to the strategy tree without dummy nodes. This shows the close relationship not only between the rules of a TA and the transitions of an AA, but also between the structures that these algorithms operate on.

This similarity of the structures is also exploited by the tableau systems framework in Chapter 7, where an S-tree that is generated by exhaustive application of the tableau rules (i. e. an S-tree *for* an input $\mathfrak{i}$) also serves as an input that is accepted by the corresponding automaton (i. e. an S-tree *compatible* with $\mathfrak{i}$).

2. Is it possible to achieve acceptable performance in practice with an automata algorithm using techniques stemming from tableau algorithms? The translation of automata into DLs in Chapter 5 allows us to perform the emptiness test for the automata with a tableau algorithm. However, our hope that the optimisations of the tableau-based reasoners could compensate for the overhead introduced by the automata construction have not materialised: empirical results show that the computation time increases exponentially in the size of the TBox (which in turn is of size exponential in the size of the input). Thus, the TBoxes generated by this translation are particularly hard to process for a TA. Upon closer examination of the structure of the TBox, it turns out that these TBoxes contain several properties that were previously observed to cause poor performance (Berardi et al., 2001), e. g. terminological cycles, inverse roles, and features.

Thus, our translation approach of alternating two-way automata into $\mathcal{FLEUI}_f$ does not lead towards a practically usable decision procedure for automata algorithms.

3. Is it possible to transfer the complexity results in either direction, i. e. is it possible to obtain a PSpace **result from an AA or an** ExpTime **result from a TA?** In order to answer the first question, we have developed the frameworks of *segmentable* and *blocking* automata in Chapter 6. These frameworks avoid the inefficiency of automata algorithms, which results from the fact that they first construct an automaton of exponential size and then test its emptiness, by interleaving the automata construction with the emptiness test and thus restricting the considered transitions to those that are relevant for the emptiness problem. By establishing a polynomial bound on the number of transitions that are kept in memory at a time, we obtain a non-deterministic algorithm requiring space polynomial in the size of the input, which, together with the theorem by Savitch (1970), gives rise to a PSpace upper bound for the emptiness test.

In the case of segmentable automata, the polynomial bound results from the fact that it is only necessary to keep one path of the run in memory at a time and that the number of (non-dummy) nodes on such a path is polynomially bounded by the size of the input. Hence, this framework is comparably easy to use, but it is restricted to logics with the finite tree model property, like $\mathcal{ALC}$. The framework of blocking automata is more general: it allows for logics requiring paths of infinite length, as long as there is a polynomial bound on the distance between two "compatible" states. The key to obtaining this result is an adaptation of the *unravelling* technique, which is used for tableau algorithms involving a blocking condition. By defining an automata algorithm for $\mathcal{SI}$ concept satisfiability w. r. t. acyclic TBoxes, we establish a PSpace upper bound for this logic.

This demonstrates the possibility of obtaining PSpace results with automata algorithms by transferring techniques known from tableau algorithms.

We have also found a method to obtain an ExpTime result from a tableau algorithm, thus answering the second question. Instead of modifying the algorithm itself, however, we developed a formalisation of tableau algorithms such that an ExpTime automata algorithm can be derived from a formalised tableau algorithm. The reason why we took this approach, which is less direct than the one for PSpace automata, is our goal to maintain the good performance of TAs in practice. Considering the negative answer to Question 2 and the significant overhead introduced by enforcing an ExpTime upper bound for a TA (Donini et al., 1996), it seems futile to aim for an algorithm that is both worst-case optimal and efficient in practice.

Therefore, we designed the *tableau system* framework (Chapter 7) in such a way that it is possible to derive from a tableau system both an ExpTime automata algorithm and a tableau algorithm that is amenable to the known optimisations for tableau algorithms and thus promises to exhibit a good performance in practice. The features defining a specific tableau system are essentially the labels appearing in the completion tree and the global memory, the completion rules, and the clash-triggers. A tableau system gives rise to an ExpTime automata algorithm if these labels are of size at most exponential in the size of the input, applicability of rules and containment of clash-triggers can be tested in exponential time, and there is a polynomial bound on the number of children of a node.

From the characteristics of the rules, we can derive an appropriate blocking condition, which ensures that the tableau algorithm induced by the tableau system terminates, although not necessarily in deterministic-exponential time. By defining a tableau system for the DL $\mathcal{SHIO}$, we obtain an ExpTime result for this logic and illustrate the usefulness of our framework.

In summary, we have established that the structural similarity of tableaus and automata is close enough to allow for the transfer of worst-case complexity results *to and fro*. The question whether it is possible to combine the desirable properties of both paradigms in such a way that a single algorithm is both worst-case optimal and efficient in practice remains open and is a possible subject for further research.

8.1 Outlook

This leads us to the question how the results presented in this thesis can be extended in the future.

Firstly, the translation approach in Chapter 5 is not optimised: it translates every possible transition into a GCI, with the consequence that the resulting TBox is of the same size as the transition function of the automaton. Thus it is an open question if it is possible to avoid such a large TBox, e. g. by developing techniques to identify redundancy within the transition function, similar to the redundancy of the transition relations in Chapter 6. The question then will be whether the detection of this redundancy can be performed efficiently enough to give a significant advantage over the naive approach.

Secondly, the frameworks for PSpace automata developed in Chapter 6 naturally can be used to obtain complexity results for new DLs or other logics by formalising them within one of the frameworks. Furthermore, by identifying properties that ensure polynomial depth but are different from segmentation and blocking, it is possible to define further frameworks that allow for capturing PSpace logics for which the corresponding automata cannot easily be shown to be segmentable or blocking.

Likewise, the Tableau Systems framework from Chapter 7 can be put to use for developing tableau algorithms and obtaining complexity results for new ExpTime logics. Moreover, tableau algorithms for $\mathcal{SHOIQ}$ (Horrocks and Sattler, 2005) and other NExpTime-complete DLs have been developed recently (see e. g. Horrocks, Kutz, and Sattler, 2006), which raises the question if a similar framework for NExpTime logics can be developed. In principle, an exponential translation of a DL input into a tree automaton with a *Rabin* acceptance condition (Rabin, 1970) would give rise to a NExpTime result since the nonemptiness problem for Rabin automata is NP-complete (Emerson and Jutla, 1988). However, $\mathcal{SHOIQ}$ does not have the tree model property, and a tree automaton therefore would have to recognise an unravelling of a model. It is unclear if this is practical with Rabin automata, whose computational power lends itself to handling fixpoints or transitive closure rather than unravellings of graphs (see Section 4.1). In order to extend the Tableau Systems framework to cover the DLs mentioned above, it will therefore be necessary to either develop a notion of tree-shaped pre-models for these logics that are recognisable by Rabin automata, or to develop automata which operate on graphs rather than trees and for which translation and emptiness test can be performed in non-deterministic exponential time.

Bibliography

Hans Albert (1991): *Traktat über kritische Vernunft*. Mohr Siebeck, Tübingen, 5th edition. (p. 16)

James Allen, Richard Fikes, and Erik Sandewall, editors (1991): *Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning (KR-91)*. Morgan Kaufmann, Los Altos. (p. 148, 150)

Aristotle (1995): *The Complete Works*, volume LXXI of *Bollingen Series*. Princeton University Press. Edited by Jonathan Barnes. (p. 16)

Franz Baader (1991): *Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles*. In Mylopoulos and Reiter (1991). (p. 27, 46)

Franz Baader (1996): *Using automata theory for characterizing the semantics of terminological cycles*. Annals of Mathematics and Artificial Intelligence **18**:175–219. (p. 26)

Franz Baader (1999): *Logic-based knowledge representation*. In *Artificial Intelligence Today, Recent Trends and Developments*, edited by Michael J. Wooldridge and Manuela M. Veloso, volume 1600 of *Lecture Notes in Computer Science*. Springer-Verlag. (p. 17, 18, 22)

Franz Baader (2003): *Terminological cycles in a description logic with existential restrictions*. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, edited by Georg Gottlob and Toby Walsh. Morgan Kaufmann, Los Altos. (p. 22)

Franz Baader, Sebastian Brandt, and Ralf Küsters (2001): *Matching under side conditions in description logics*. In Nebel (2001). (p. 22)

Franz Baader, Sebastian Brandt, and Carsten Lutz (2005): *Pushing the $\mathcal{EL}$ envelope*. In Kaelbling and Saffiotti (2005). (p. 22, 26, 71)

Franz Baader, Martin Buchheit, and Bernhard Hollunder (1996): *Cardinality restrictions on concepts*. Artificial Intelligence Journal **88**(1–2):195–213. (p. 41, 43, 134)

Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors (2003a): *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press. (p. 21, 145, 153)

Franz Baader, Jan Hladik, Carsten Lutz, and Frank Wolter (2003b): *From tableaux to automata for description logics*. In *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003)*, edited by Moshe Vardi and Andrei Voronkov, volume 2850 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 107)

Franz Baader, Jan Hladik, Carsten Lutz, and Frank Wolter (2003c): *From tableaux to automata for description logics*. Fundamenta Informaticae **57**:1–33. (p. 107)

Franz Baader, Jan Hladik, and Rafael Peñaloza (2006): *PSPACE automata with blocking for description logics*. LTCS-Report LTCS-06-04, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Germany. Available from `http://lat.inf.tu-dresden.de/research/reports.html`. (p. 84)

Franz Baader, Jan Hladik, and Rafael Peñaloza (2007a): *Blocking automata for PSPACE DLs*. In *Proceedings of the 2007 Description Logic Workshop (DL 2007)*, edited by Diego Calvanese, Enrico Franconi, Volker Haarslev, Domenico Lembo, Boris Motik, Sergio Tessaris, and Anni-Yasmin Turhan, volume 250 of *CEUR Workshop Proceedings*. Available from `ceur-ws.org`. (p. 84)

Franz Baader, Jan Hladik, and Rafael Peñaloza (2008): *$\mathcal{SI}$! Automata can show PSPACE results for description logics*. Information and Computation, Special Issue: First International Conference on Language and Automata Theory and Applications (LATA'07). To appear. (p. 84)

Franz Baader and Bernhard Hollunder (1991a): *$\mathcal{KRIS}$: Knowledge Representation and Inference System*. SIGART Bulletin **2**(3):8–14. (p. 30)

Franz Baader and Bernhard Hollunder (1991b): *A terminological knowledge representation system with complete inference algorithm*. In *Proceedings of the Workshop on Processing Declarative Knowledge (PDK'91)*, edited by Harold Boley and Michael M. Richter, volume 567 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 30)

Franz Baader, Ralf Küsters, and Ralf Molitor (1998): *Computing least common subsumers in description logics with existential restrictions*. LTCS-Report LTCS-98-09, LuFG Theoretical Computer Science, RWTH Aachen, Germany. Available from `lat.inf.tu-dresden.de/research/reports.html`. (p. 26)

Franz Baader, Ralf Küsters, and Ralf Molitor (1999): *Computing least common subsumers in description logics with existential restrictions*. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, edited by Thomas Dean. Morgan Kaufmann, Los Altos. (p. 22, 25)

Franz Baader, Carsten Lutz, and Boontawee Suntisrivaraporn (2007b): *Is tractable reasoning in extensions of the description logic $\mathcal{EL}$ useful in practice?* Journal of Logic, Language and Information, Special Issue: Methods for Modalities (M4M-4). To appear. (p. 22)

Franz Baader and Werner Nutt (2003): *Reasoning algorithms.* In Baader et al. (2003a), chapter 2.3. (p. 34)

Franz Baader and Ulrike Sattler (2001): *An overview of tableau algorithms for description logics.* Studia Logica **69**:5–40. (p. 25, 34, 35, 125)

Franz Baader and Stephan Tobies (2001): *The inverse method implements the automata approach for modal satisfiability.* In Goré, Leitsch, and Nipkow (2001). (p. 12)

Andrew B. Baker (1995): *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* Ph.D. thesis, University of Oregon. (p. 39)

Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn A. Stein (2004): *OWL web ontology language reference.* `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`. (p. 22)

Mordechai Ben-Ari, Joseph Y. Halpern, and Amir Pnueli (1982): *Deterministic propositional dynamic logic: Finite models, complexity, and completeness.* Journal of Computer and System Science **25**:402–417. (p. 71)

Daniela Berardi, Diego Calvanese, and Giuseppe de Giacomo (2001): *Reasoning on UML class diagrams using description logic based systems.* In *Proceedings of the KI'2001 Workshop on Applications of Description Logics*, edited by Günther Görz, volume 44 of *CEUR Workshop Proceedings.* Available from `ceur-ws.org`. (p. 79, 140)

Evert Willem Beth (1955): *Semantic entailment and formal derivability.* Mededelingen der Koninklijke Nederlandse Adademie van Wetenschappen, Nieuwe Reeks **18**(13):309–342. (p. 33)

Evert Willem Beth (1959): *The Foundations of Mathematics.* North-Holland Publ. Co., Amsterdam. (p. 34)

Patrick Blackburn, Maarten de Rijke, and Yde Venema (2001): *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press. (p. 21, 27, 110)

Daniel G. Bobrow and Terry Winograd (1977): *An overview of KRL, a Knowledge Representation Language.* Cognitive Science **1**(1):3–46. Republished in Brachman and Levesque (1985). (p. 20)

Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick (1989): *CLASSIC: A structural data model for objects.* In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, edited by James Clifford, Bruce G. Lindsay, and David Maier. ACM Press and Addison Wesley. (p. 30)

Ronald J. Brachman (1979): *On the epistemological status of semantic networks.* In *Associative Networks*, edited by Nicholas V. Findler, pages 3–50. Academic Press. Republished in Brachman and Levesque (1985). (p. 19)

Ronald J. Brachman (1983): *What IS-A is and isn't: An analysis of taxonomic links in semantic networks.* IEEE Computer **16**(10):30–36. (p. 19)

Ronald J. Brachman (1985): *"I lied about the trees".* AI Magazine **6**(3):80–93. (p. 19)

Ronald J. Brachman and Hector J. Levesque, editors (1985): *Readings in Knowledge Representation.* Morgan Kaufmann, Los Altos. (p. 145, 146, 149, 152, 154)

Ronald J. Brachman and Hector J. Levesque (2004): *Knowledge Representation and Reasoning.* Morgan Kaufmann, Los Altos. (p. 17)

Ronald J. Brachman and James G. Schmolze (1985): *An overview of the KL-ONE knowledge representation system.* Cognitive Science **9**(2):171–216. (p. 21)

Sebastian Brandt (2004): *Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else?* In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-2004)*, edited by Ramon López de Mantáras and Lorenza Saitta. IOS Press. (p. 22)

Sebastian Brandt, Ralf Küsters, and Anni-Yasmin Turhan (2002): *Approximation and difference in description logics.* In *Proceedings of the Eighth International Conference on the Principles of Knowledge Representation and Reasoning (KR-02)*, edited by Dieter Fensel, Fausto Giunchiglia, Deborah McGuiness, and Mary-Anne Williams. Morgan Kaufmann, Los Altos. (p. 22)

Martin Buchheit, Francesco M. Donini, and Andrea Schaerf (1993): *Decidable reasoning in terminological knowledge representation systems.* Journal of Artificial Intelligence Research **1**:109–138. (p. 41)

Julius Richard Büchi (1960): *On a decision method in restricted second order arithmetic.* In *Proceedings of the Internatinal Congress on Logic, Methodology and Philosophy of Science*, edited by Ernest Nagel, Patrick Suppes, and Alfred Tarski. Stanford University Press. (p. 46)

Diego Calvanese (1996): *Reasoning with inclusion axioms in description logics: Algorithms and complexity.* In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, edited by Wolfgang Wahlster. John Wiley & Sons. (p. 71)

Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini (1999): *Reasoning in expressive description logics with fixpoints based on automata on infinite trees.* In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, edited by Thomas Dean. Morgan Kaufmann, Los Altos. (p. 46, 47, 59, 71)

Diego Calvanese, Giuseppe de Giacomo, and Maurizio Lenzerini (2002): *2ATAs make DLs easy.* In *Proceedings of the 2002 Description Logic Workshop (DL 2002)*, edited by Ian Horrocks and Sergio Tessaris, volume 53 of *CEUR Proceedings*. Available from `ceur-ws.org`. (p. 59, 62, 140)

William W. Cohen, Alex Borgida, and Haym Hirsh (1992): *Computing least common subsumers in description logics.* In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, edited by William Swartout. AAAI Press/The MIT Press. (p. 22)

Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors (1998): *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning (KR-98)*. Morgan Kaufmann, Los Altos. (p. 148, 150)

Allan M. Collins and Elizabeth F. Loftus (1975): *A spreading-activation theory of semantic processing.* Psychological Review **6**(82):407–428. (p. 19)

Marcello D'Agostino, Dov M. Gabbay, Reiner Hähnle, and Joachim Posegga, editors (1999): *Handbook of Tableau Methods.* Kluwer Academic Publishers, Dordrecht. (p. 33, 148)

Martin Davis, George Logemann, and Donald Loveland (1962): *A machine program for theorem-proving.* Communications of the ACM **5**(7):394–397. (p. 13, 40)

Martin Davis and Hilary Putnam (1960): *A computing procedure for quantification theory.* Journal of the ACM **7**(3):201–215. (p. 13)

Giuseppe De Giacomo (1995): *Decidability of Class-Based Knowledge Representation Formalisms.* Ph.D. thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza". (p. 23)

Giuseppe De Giacomo (1996): *Eliminating "converse" from Converse* **PDL** . Journal of Logic, Language and Information **5**(2):193–208. (p. 28)

Giuseppe De Giacomo and Maurizio Lenzerini (1994a): *Boosting the correspondence between description logics and propositional dynamic logics.* In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press/The MIT Press. (p. 13)

Giuseppe De Giacomo and Maurizio Lenzerini (1994b): *Description logics with inverse roles, functional restrictions, and n-ary relations.* In *Proceedings of the 4th European Conference on Logics in Artificial Intelligence (JELIA-94)*, edited by Craig MacNish, David Pearce, and Luis M. Pereira, volume 838 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13)

Yu Ding and Volker Haarslev (2007): *A procedure for description logic* $\mathcal{ALCFI}$. In Olivetti (2007). (p. 44)

Patrick Doherty, John Mylopoulos, and Christopher Welty, editors (2006): *Proceedings of the Tenth International Conference on the Principles of Knowledge Representation and Reasoning (KR-06)*. AAAI Press. (p. 148, 150)

Francesco Donini, Giuseppe De Giacomo, and Fabio Massacci (1996): *EXPTIME tableaux for $\mathcal{ALC}$*. In *Proceedings of the 1996 Description Logic Workshop (DL'96)*, edited by Lin Padgham, Enrico Franconi, Manfred Gehrke, Deborah L. McGuinness, and Peter F. Patel-Schneider, number WS-96-05 in AAAI Technical Reports. AAAI Press/The MIT Press. (p. 43, 44, 141)

Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Werner Nutt (1991): *The complexity of concept languages*. In Allen, Fikes, and Sandewall (1991). (p. 26)

E. Allen Emerson and Charanjit S. Jutla (1988): *The complexity of tree automata and logics of programs*. In *Proceedings of the Twenty-Ninth Annual Symposium on the Foundations of Computer Science (FOCS-88)*. IEEE Computer Society Press. (p. 47, 142)

Michael J. Fischer and Richard E. Ladner (1979): *Propositional dynamic logic of regular programs*. Journal of Computer and System Science **18**:194–211. (p. 27)

Melvin Fitting (1999): *Introduction to tableau methods*. In D'Agostino et al. (1999), chapter 1. (p. 34)

Harald Ganzinger, editor (1999): *Proceedings of the 16th Conference on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 155, 156)

Gerhard Gentzen (1935): *Untersuchungen über das logische Schließen*. Mathematische Zeitschrift **39**:176–210, 405–431. (p. 33)

Silvio Ghilardi, Carsten Lutz, and Frank Wolter (2006): *Did I damage my ontology? A case for conservative extensions in description logics*. In Doherty, Mylopoulos, and Welty (2006). (p. 22)

Enrico Giunchiglia, Fausto Giunchiglia, Roberto Sebastiani, and Armando Tacchella (1998): *More evaluation of decision procedures for modal logics*. In Cohn, Schubert, and Shapiro (1998). (p. 13)

Fausto Giunchiglia and Roberto Sebastiani (1996): *A SAT-based decision procedure for $\mathcal{ALC}$*. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR-96)*, edited by Luigia C. Aiello, John Doyle, and Stuart C. Shapiro. Morgan Kaufmann, Los Altos. (p. 13, 31)

Birte Glimm, Ian Horrocks, Carsten Lutz, and Ulrike Sattler (2007): *Conjunctive query answering for the description logic $\mathcal{SHIQ}$*. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, edited by Manuela M. Veloso. AAAI Press/The MIT Press. (p. 22)

Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors (2001): *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR-01)*, volume 2083 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 145, 149)

Rajeev P. Goré and Linh Nguyen (2007): *EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies*. In Olivetti (2007). (p. 44)

Volker Haarslev and Ralf Möller (2001a): *High performance reasoning with very large knowledge bases: A practical case study*. In Nebel (2001). (p. 30)

Volker Haarslev and Ralf Möller (2001b): *RACER system description*. In Goré et al. (2001). (p. 30, 65)

Volker Haarslev, Ralf Möller, and Michael Wessel (2005a): *RacerPro Reference Manual Version 1.9*. Racer Systems GmbH & Co. KG. Available from: `http://www.racer-systems.com/products/racerpro/reference-manual-1-9.pdf`. (p. 30)

Volker Haarslev, Ralf Möller, and Ralf Wessel (2005b): *Description logic inference technology: Lessions learned in the trenches*. In *Proceedings of the 2005 Description Logic Workshop (DL 2005)*, edited by Ian Horrocks, Ulrike Sattler, and Frank Wolter, volume 147 of *CEUR Proceedings*. Available from `ceur-ws.org`. (p. 30)

Patrick J. Hayes (1974): *Some problems and non-problems in representation theory*. In *Proceedings of the AISB Summer Conference*. University of Sussex. (p. 20)

Patrick J. Hayes (1979): *The logic of frames*. In *Frame Conceptions and Text Understanding*, edited by Dieter Metzing, pages 46–61. Walter de Gruyter and Co. Republished in Brachman and Levesque (1985). (p. 20)

K. Jaakko J. Hintikka (1955): *Form and content in quantification theory*. Acta Philosophica Fennica **8**:8–55. (p. 34)

Jan Hladik (2002): *Implementation and optimisation of a tableau algorithm for the guarded fragment*. In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2002)*, edited by Uwe Egly and Christian G. Fermüller, volume 2381 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 40, 78)

Jan Hladik (2003): *Reasoning about nominals with FaCT and RACER*. In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, edited by Diego Calvanese, Giuseppe di Giacomo, and Enrico Franconi, volume 81 of *CEUR Workshop Proceedings*. Available from `ceur-ws.org`. (p. 65)

Jan Hladik (2004): *A tableau system for the description logic $\mathcal{SHIO}$*. In *Contributions to the Doctoral Programme of IJCAR 2004*, edited by Ulrike Sattler, volume 106 of *CEUR Workshop Proceedings*. Available from `ceur-ws.org`. (p. 107)

Jan Hladik and Jörg Model (2004): *Tableau systems for $\mathcal{SHIO}$ and $\mathcal{SHIQ}$.* In *Proceedings of the 2004 Description Logic Workshop (DL 2004)*, edited by Volker Haarslev and Ralf Möller, volume 104 of *CEUR Workshop Proceedings*. Available from `ceur-ws.org`. (p. 107)

Jan Hladik and Rafael Peñaloza (2006): *PSPACE automata for description logics.* In *Proceedings of the 2006 Description Logic Workshop (DL 2006)*, edited by Bijan Parsia, Ulrike Sattler, and David Toman, volume 189 of *CEUR Workshop Proceedings*. Available from `ceur-ws.org`. (p. 84)

Jan Hladik and Ulrike Sattler (2003): *A translation of looping alternating automata into description logics.* In *Proceedings of the 19th Conference on Automated Deduction (CADE-19)*, edited by Franz Baader, volume 2741 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 65)

Bernhard Hollunder and Franz Baader (1991): *Qualifying number restrictions in concept languages.* In Allen et al. (1991). (p. 116)

Ian Horrocks (1997): *Optimising Tableaux Decision Procedures for Description Logics.* Ph.D. thesis, University of Manchester. (p. 30, 39, 78, 79)

Ian Horrocks (1998a): *The FaCT system.* In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98)*, edited by Harrie de Swart, volume 1397 of *Lecture Notes in Artificial Intelligence*, pages 307–312. Springer-Verlag. (p. 30)

Ian Horrocks (1998b): *Using an expressive description logic: FaCT or fiction?* In Cohn et al. (1998). (p. 65)

Ian Horrocks, Oliver Kutz, and Ulrike Sattler (2006): *The even more irresistible $\mathcal{SROIQ}$.* In Doherty et al. (2006). (p. 142)

Ian Horrocks and Peter F. Patel-Schneider (1999): *Optimizing description logic subsumption.* Journal of Logic and Computation **9**(3):267–293. (p. 31, 40)

Ian Horrocks and Ulrike Sattler (1999): *A description logic with transitive and inverse roles and role hierarchies.* Journal of Logic and Computation **9**(3):385–410. (p. 27, 125, 129)

Ian Horrocks and Ulrike Sattler (2001): *Ontology reasoning in the $\mathcal{SHOQ}(D)$ description logic.* In Nebel (2001). (p. 28, 35)

Ian Horrocks and Ulrike Sattler (2002): *Optimised reasoning for $\mathcal{SHIQ}$.* In *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-2002)*, edited by Frank van Harmelen. IOS Press. (p. 44)

Ian Horrocks and Ulrike Sattler (2005): *A tableaux decision procedure for $\mathcal{SHOIQ}$.* In Kaelbling and Saffiotti (2005). (p. 22, 30, 142)

Ian Horrocks, Ulrike Sattler, and Stefan Tobies (2000a): *Practical reasoning for very expressive description logics.* Logic Journal of the IGPL **8**(3):239–264. (p. 27, 28, 41, 44, 65, 129, 133, 134, 138)

Ian Horrocks, Ulrike Sattler, and Stephan Tobies (1998): *A PSPACE-algorithm for deciding $\mathcal{ALCNI}_{R^+}$-satisfiability.* LTCS-Report LTCS-98-08, LuFg Theoretical Computer Science, RWTH Aachen, Germany. Available from `lat.inf.tu-dresden.de/research/reports.html`. (p. 94)

Ian Horrocks, Ulrike Sattler, and Stephan Tobies (1999): *Practical reasoning for expressive description logics.* In *Proceedings of the Sixth International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, edited by Harald Ganzinger, David McAllester, and Andrei Voronkov, number 1705 in Lecture Notes in Artificial Intelligence. Springer-Verlag. (p. 27, 84, 92, 125, 137)

Ian Horrocks, Ulrike Sattler, and Stephan Tobies (2000b): *Reasoning with individuals for the description logic $\mathcal{SHIQ}$.* In *Proceedings of the 17th Conference on Automated Deduction (CADE-17)*, edited by David MacAllester, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 25)

Ullrich Hustadt, Boris Motik, and Ulrike Sattler (2004): *Reducing $\mathcal{SHIQ}^-$ description logic to disjunctive datalog programs.* In *Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR-04)*, edited by Didier Dubois, Christopher Welty, and Mary-Anne Williams. Morgan Kaufmann, Los Altos. (p. 13, 31)

Ullrich Hustadt, Boris Motik, and Ulrike Sattler (2007): *Reasoning in description logics by a reduction to disjunctive datalog.* Journal of Automated Reasoning **39**(3):351–384. (p. 13)

Ulrich Hustadt and Renate A. Schmidt (1997): *On evaluating decision procedures for modal logic.* In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, edited by Martha E. Pollack. Morgan Kaufmann, Los Altos. (p. 31)

Ulrich Hustadt and Renate A. Schmidt (2000): MSPASS*: Modal reasoning by translation and first-order resolution.* In *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2000)*, edited by R. Dyckhoff, volume 1847 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13, 31)

Leslie Pack Kaelbling and Alessandro Saffiotti, editors (2005): *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05).* Professional Book Center. (p. 143, 150)

Yevgeny Kazakov and Hans de Nivelle (2003): *Subsumption of concepts in $\mathcal{FL}_0$ for (cyclic) terminologies with respect to descriptive semantics is PSPACE-complete.* In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, edited by Diego

Calvanese, Giuseppe di Giacomo, and Enrico Franconi, volume 81 of *CEUR Proceedings*. Available from `ceur-ws.org`. (p. 26)

Stephen C. Kleene (1956): *Representation of events in nerve nets and finite automata*. In *Automata Studies*, edited by Claude E. Shannon and John McCarthy. Princeton University Press, Princeton, USA. (p. 45, 46)

Orna Kupferman and Moshe Y. Vardi (1998): *Weak alternating automata and tree automata emptiness*. In *Proceedings of the Thirtieth ACM SIGACT Symposium on Theory of Computing (STOC-98)*. ACM Press and Addison Wesley. (p. 47)

Gerhard Lakemeyer and Bernhard Nebel (1994): *Foundations of Knowledge Representation and Reasoning*, volume 810 of *Lecture Notes in Artificial Intelligence*, chapter 1, pages 1–12. Springer-Verlag. (p. 17)

Hector J. Levesque and Ronald J. Brachman (1985): *A fundamental tradeoff in knowledge representation and reasoning*. In Brachman and Levesque (1985). (p. 17, 18, 21)

Carsten Lutz (2002): *Adding numbers to the $\mathcal{SHIQ}$ description logic—First results*. In *Proceedings of the Eighth International Conference on the Principles of Knowledge Representation and Reasoning (KR-02)*, edited by Dieter Fensel, Fausto Giunchiglia, Deborah McGuinnes, and Mary-Anne Williams. Morgan Kaufmann, Los Altos. (p. 49)

Carsten Lutz and Ulrike Sattler (2000): *Mary likes all cats*. In *Proceedings of the 2000 Description Logic Workshop (DL 2000)*, edited by Franz Baader and Ulrike Sattler, volume 33 of *CEUR Proceedings*. Available from `ceur-ws.org`. (p. 49)

Robert MacGregor and Raymond Bates (1987): *The LOOM knowledge representation language*. Technical Report ISI-RS-87-188, USC Information Sciences Institute, Marina del Rey, California, USA. (p. 30)

David A. McAllester, Robert Givan, Carl Witty, and Dexter Kozen (1996): *Tarskian set constraints*. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS-96)*. IEEE Computer Society Press. (p. 27)

John McCarthy (1958): *Programs with common sense*. In *Mechanisation of Thought Processes*, volume 1, pages 77–84. National Physical Laboratory, London. Reprinted in Brachman and Levesque (1985). (p. 18)

Drew McDermott (1986): *Tarskian semantics, or no notation without denotation*. In *Readings in Natural Language Processing*, edited by Barbara J. Grosz, Karen Sparck-Jones, and Bonnie L. Webber. Morgan Kaufmann, Los Altos. (p. 20)

Marvin Minsky (1975): *A framework for representing knowledge*. In *The Psychology of Computer Vision*, edited by Patrick H. Winston. McGraw-Hill, New York. A shorter version was republished in Brachman and Levesque (1985). (p. 19, 20)

Boris Motik and Ulrike Sattler (2006): *A comparison of reasoning techniques for querying large description logic ABoxes*. In *Proceedings of the 13th International Conference on Logic for Programming and Automated Reasoning (LPAR 2006)*, edited by Miki Hermann and Andrei Voronkov, volume 4246 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13, 31)

Boris Motik, Rob Shearer, and Ian Horrocks (2007): *Optimized reasoning in description logics using hypertableaux*. In *Proceedings of the 21st Conference on Automated Deduction (CADE-19)*, edited by Frank Pfenning, volume 4603 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13, 31)

David E. Muller and Paul E. Schupp (1987): *Alternating automata on infinite trees*. Theoretical Computer Science **54**:267–276. (p. 53)

David E. Muller and Paul E. Schupp (1995): *Simulating alternating tree automata by nondeterministic automata: new results and new proofs of the theorems of Rabin, McNaughton and Safra*. Theoretical Computer Science **141**:69–107. (p. 62)

John Mylopoulos and Raymond Reiter, editors (1991): *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*. Morgan Kaufmann, Los Altos. (p. 143, 155)

Daniele Nardi and Ronald J. Brachman (2003): *An introduction to description logics*. In Baader et al. (2003a), chapter 1. (p. 21)

Bernhard Nebel (1990): *Terminological reasoning is inherently intractable*. Artificial Intelligence Journal **43**:235–249. (p. 21, 26, 29)

Bernhard Nebel, editor (2001): *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*. Morgan Kaufmann, Los Altos. (p. 143, 149, 150, 154)

Nicola Olivetti, editor (2007): *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2007)*, volume 4548 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 147, 149)

Bernd Owsnicki-Klewe, Kai von Luck, and Bernhard Nebel (1995): *Wissensrepräsentation und Logik - Eine Einführung*. In *Einführung in die Künstliche Intelligenz*, edited by Günther Görz, chapter 1.1. Addison-Wesley, 2nd edition. (p. 17)

Leszek Pacholski, Wieslaw Szwast, and Lidia Tendera (1997): *Complexity of two-variable logic with counting*. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science (LICS-97)*. IEEE Computer Society Press. (p. 29)

Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi (2002): *BDD-based decision procedures for* **K**. In *Proceedings of the 18th Conference on Automated Deduction (CADE-18)*, edited by Andrei Voronkov, volume 2392 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13)

Guoqiang Pan, Ulrike Sattler, and Moshe Y. Vardi (2006): *BDD-based decision procedures for the modal logic* **K**. Journal of Applied Non-Classical Logics **16**(1–2):169–208. (p. 12, 31)

Guoqiang Pan and Moshe Y. Vardi (2003): *Optimizing a BDD-based modal solver*. In *Proceedings of the 19th Conference on Automated Deduction (CADE-19)*, edited by Franz Baader, volume 2741 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 13)

Peter F. Patel-Schneider, Deborah L. McGuiness, Ronald J. Brachman, Lori Alperin Resnick, and Alexander Borgida (1991): *The CLASSIC knowledge representation system: Guiding principles and implementation rational*. SIGART Bulletin **2**(3):108–113. (p. 30)

Dominique Perrin (1990): *Finite automata*. In *Handbook of Theoretical Computer Science*, edited by Jan van Leeuwen, volume B. Elsevier Science Publishers (North-Holland), Amsterdam. (p. 45)

Plato (1997): *Complete Works*. Hackett Publishing Company. Edited by John M. Cooper. (p. 15)

M. Ross Quillian (1967): *Word concepts: A theory and simulation of some basic capabilities*. Behavioral Science **12**:410–430. Republished in Brachman and Levesque (1985). (p. 19)

Michael O. Rabin (1969): *Decidability of second-order theories and automata on infinite trees*. Transactions of the American Mathematical Society **141**:1–35. (p. 46)

Michael O. Rabin (1970): *Weakly definable relations and special automata*. In *Proceedings of the Symposium on Mathematical Logic and Foundations of Set Theory*, edited by Yehoshua Bar-Hillel. North-Holland Publ. Co., Amsterdam. (p. 46, 142)

Raymond Reiter (1978): *On reasoning by default*. In *Proceedings of the 2nd Symposium on Theoretical Issues in Natural Language Processing*. Association for Computational Linguistics, Morristown, NJ, USA. (p. 20)

Stuart Russell and Peter Norvig (2002): *Logical agents*. In *Artificial intelligence: A Modern Approach*, chapter 7. Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition. (p. 18)

Ulrike Sattler (1996): *A concept language extended with different kinds of transitive roles*. In *Proceedings of the 20th German Annual Conf. on Artificial Intelligence (KI'96)*, edited by Günter Görz and Steffen Hölldobler, number 1137 in Lecture Notes in Artificial Intelligence. Springer-Verlag. (p. 23)

Ulrike Sattler and Moshe Y. Vardi (2001): *The hybrid μ-calculus*. In Nebel (2001). (p. 46, 60, 73, 76)

Walter J. Savitch (1970): *Relationship between nondeterministic and deterministic tape complexities.* Journal of Computer and System Science **4**:177–192. (p. 9, 84, 92, 140)

Klaus Schild (1991): *A correspondence theory for terminological logics: Preliminary report.* In Mylopoulos and Reiter (1991). (p. 13, 21, 23, 27)

Klaus Schild (1994): *Terminological cycles and the propositional μ-calculus.* In *Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning (KR-94)*, edited by Jon Doyle, Erik Sandewall, and Pietro Torasso. Morgan Kaufmann, Los Altos. (p. 133)

Manfred Schmidt-Schauß (1989): *Subsumption in KL-ONE is undecidable.* In *Proceedings of the First International Conference on the Principles of Knowledge Representation and Reasoning (KR-89)*, edited by Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter. Morgan Kaufmann, Los Altos. (p. 21)

Manfred Schmidt-Schauß and Gert Smolka (1991): *Attributive concept descriptions with complements.* Artificial Intelligence Journal **48**(1):1–26. (p. 21, 26, 27, 39, 92)

Evren Sirin and Bijan Parsia (2004): *Pellet: an OWL-DL reasoner.* In *Proceedings of the 2004 Description Logic Workshop (DL 2004)*, edited by Volker Haarslev and Ralf Möller, volume 104 of *CEUR Proceedings*. Available from `ceur-ws.org`. (p. 30)

Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz (2007): *Pellet: a practical OWL-DL reasoner.* Journal of Web Semantics **5**(2):51–53. (p. 30)

Brian C. Smith (1982): *Reflection and Semantics in a Procedural Language.* Ph.D. thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, USA. Technical Report No. MIT/LCS/TR-272. (p. 17)

Raymond M. Smullyan (1968): *First Order Logic.* Springer-Verlag, Berlin (Germany). (p. 34)

Stephan Tobies (1999): *A PSPACE algorithm for graded modal logic.* In Ganzinger (1999). (p. 116)

Stephan Tobies (2000): *The complexity of reasoning with cardinality restrictions and nominals in expressive description logics.* Journal of Artificial Intelligence Research **12**:199–217. (p. 29)

Stephan Tobies (2001): *Complexity Results and Practical Algorithms for Logics in Knowledge Representation.* Ph.D. thesis, RWTH Aachen, Germany. (p. 129)

Dmitry Tsarkov and Ian Horrocks (2006): *FaCT++ description logic reasoner: System description.* In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006)*, edited by Ulrich Furbach and Natarajan Shankar, volume 4130 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 30)

Moshe Y. Vardi (1989): *A note on the reduction of two-way automata to one-way automata*. Information Processing Letters **30**(5):261–264. (p. 54)

Moshe Y. Vardi (1997): *Alternating automata: Unifying truth and validity checking for temporal logics*. In *Proceedings of the 14th Conference on Automated Deduction (CADE-97)*, edited by William McCune, volume 1249 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag. (p. 62)

Moshe Y. Vardi (1998): *Reasoning about the past with two-way automata*. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, edited by Kim G. Larsen, Sven Skyum, and Glynn Winskel, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag. (p. 47, 55)

Moshe Y. Vardi and Pierre Wolper (1986): *Automata-theoretic techniques for modal logics of programs*. Journal of Computer and System Science **32**:183–221. (p. 46, 47, 124)

Andrei Voronkov (1999): KK*: A theorem prover for* **K**. In Ganzinger (1999). (p. 12, 30)

Andrei Voronkov (2001): *How to optimize proof-search in modal logics: new methods of proving redundancy criteria for sequent calculi*. ACM transactions on computational logic **2**(2). (p. 12, 30)

Christoph Weidenbach (1999): *SPASS: Combining superposition, sorts and splitting*. In *Handbook of Automated Reasoning*, edited by J. Alan Robinson and Andrei Voronkov, chapter 27. Elsevier Science Publishers (North-Holland), Amsterdam. (p. 13)

Christoph Weidenbach, Bijan Afshordel, Uwe Brahm, Christian Cohrs, Thorsten Engel, Enno Keen, Christian Theobalt, and Dalibor Topić (1999): *System description: SPASS version 1.0.0*. In Ganzinger (1999). (p. 13)

William A. Woods and James G. Schmolze (1990): *The KL-ONE family*. Technical Report TR-20-90, Aiken Computation Laboratory, Harvard University, Cambridge (MA, USA). Published in a special issue of *Computers & Mathematics with Applications*, Volume 23, Number 2–9. (p. 21)

Index