Putting ABox Updates into Action

Conrad Drescher, Hongkai Liu, Franz Baader, Steffen Guhlemann, Uwe Petersohn, Peter Steinke, Michael Thielscher

Department of Computer Science,
Dresden University of Technology
Nöthnitzer Str. 46, 01187 Dresden, Germany

Abstract. When trying to apply recently developed approaches for updating Description Logic ABoxes in the context of an action programming language, one encounters two problems. First, updates generate so-called *Boolean* ABoxes, which cannot be handled by traditional Description Logic reasoners. Second, iterated update operations result in *very large* Boolean ABoxes, which, however, contain a huge amount of redundant information. In this paper, we address both issues from a practical point of view.

1 Introduction

Agent programming languages such as Golog [1] and Flux [2] employ actions whose effects are defined in a logic-based calculus to describe and implement the behaviour of intelligent agents. In the so-called progression approach, the agent starts with a (possibly incomplete) description of the initial state of the world. When an action is performed, it updates this description to take into account the effects of this action. Reasoning about the description of the current state of the world is then, for example, used in the control structures of the agent program to decide which action to apply. The calculi underlying Golog and Flux (situation calculus and fluent calculus, respectively) employ full first-order predicate logic, which makes the computation of exact updates as well as the use of decision procedures for reasoning about descriptions of the state of the world impossible. To overcome this problem, recent papers [3, 4] have proposed to employ a decidable Description Logic (DL) [5] in place of full first-order predicate logic. In particular, states of the world are then described using a DL ABox. In [4], a method for updating DL ABoxes has been developed, and in [6] it was shown that this notion of an update conforms with the semantics employed by Golog and Flux.

In practice, however, there are two obstacles towards employing the update approach from [4] in the context of agent programs. First, using the update procedures in the form described in [4] quickly leads to unmanageably large ABoxes. However, there is quite some room for optimizations since the updated ABoxes contain a lot of redundant information. The second problem is that the updated ABoxes are so-called Boolean ABoxes, which cannot be directly handled by traditional DL reasoners. The main contributions of this paper are,

on the one hand, that we propose and evaluate different optimization approaches for computing more concise updated ABoxes. On the other hand, we compare different approaches for reasoning with Boolean ABoxes, among them one based on the DPLL(T) approach.

The rest of this paper is organized as follows. In Section 2, we recall the basic notions for DLs and ABox updates. In Sections 3 we present optimizations that enable the construction of more concise updated ABoxes, and in Section 4 we discuss reasoning with Boolean ABoxes. In Section 5, the approaches introduced in the previous two sections are empirically evaluated. This paper is also available as a longer technical report [7].

2 Preliminaries

In DLs, knowledge is represented with the help of concepts (unary predicates) and roles (binary predicates). Complex concepts and roles are inductively defined starting with a set $\mathsf{N_C}$ of *concept names*, a set $\mathsf{N_R}$ of *role names*, and a set $\mathsf{N_I}$ of *individual names*. The expressiveness of a DL is determined by the set of available *constructors* to build *concepts* and *roles*. The concept and role constructors of the DLs $\mathcal{ALCO}^{@}$ and $\mathcal{ALCO}^{+}$ that form the base of our work on ABox update are shown in Table 1, where C, D are concepts, q, r are roles, and a, b are individual names. The DL that allows only for negation, conjunction, disjunction, and universal and existential restrictions is called $\mathcal{ALC}$. By adding nominals $\mathcal{O}$, we obtain $\mathcal{ALCO}$, which is extended to $\mathcal{ALCO}^{@}$ by the @-constructor from hybrid logic [8], and to $\mathcal{ALCO}^{+}$ by the Boolean constructors on roles and the nominal role [4]. We will use $\top$ ($\bot$) to denote arbitrary tautological (unsatisfiable) concepts and roles. By $\mathsf{sub}(\phi)$ we denote the set of all subconcepts and subroles of a concept or role ϕ, respectively.

Name	Syntax	Semantics
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
universal restriction	$\forall r.C$	$\{x \mid \forall y.((x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}})\}$
existential restriction	$\exists r.C$	$\{x \mid \exists y.((x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}})\}$
nominal	$\{a\}$	$\{a^{\mathcal{I}}\}$
@ constructor	$@_a C$	$\Delta^{\mathcal{I}}$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$, and $\emptyset$ otherwise
role negation	$\neg r$	$(\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}) \setminus r^{\mathcal{I}}$
role conjunction	$q \sqcap r$	$q^{\mathcal{I}} \cap r^{\mathcal{I}}$
role disjunction	$q \sqcup r$	$q^{\mathcal{I}} \cup r^{\mathcal{I}}$
nominal role	$\{(a,b)\}$	$\{(a^{\mathcal{I}}, b^{\mathcal{I}})\}$

Table 1. Syntax and semantics of $\mathcal{ALCO}^{@}$ and $\mathcal{ALCO}^{+}$.

The semantics of concepts and roles is given via *interpretations* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. The *domain* $\Delta^{\mathcal{I}}$ is a non-empty set and the *interpretation function* $\cdot^{\mathcal{I}}$ maps each concept name $A \in \mathsf{N_C}$ to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, each role name $r \in \mathsf{N_R}$ to a binary relation $r^{\mathcal{I}}$ on $\Delta^{\mathcal{I}}$, and each individual name $a \in \mathsf{N_I}$ to an individual $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The interpretation function $\cdot^{\mathcal{I}}$ is inductively extended to complex concepts and roles as shown in Table 1.

An *ABox assertion* is of the form $C(a)$, $r(a, b)$, or $\neg r(a, b)$ with r a role, C a concept and a, b individual names. A *classical ABox*, or an *ABox* for short, is a finite conjunction of ABox assertions. A *Boolean ABox* is a Boolean combination of ABox assertions. For convenience we will also sometimes represent classical and Boolean ABoxes as finite sets of assertions by breaking the toplevel conjunctions. An interpretation $\mathcal{I}$ is a *model* of an assertion $C(a)$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$. $\mathcal{I}$ is a model of an assertion $r(a, b)$ (resp. $\neg r(a, b)$) if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$ (resp. $(a^{\mathcal{I}}, b^{\mathcal{I}}) \notin r^{\mathcal{I}}$). A model of a (Boolean) ABox is defined in the obvious way. We use $M(\mathcal{A})$ to denote the set of models of a Boolean ABox $\mathcal{A}$. A (Boolean) ABox $\mathcal{A}$ is *consistent* if $M(\mathcal{A}) \neq \emptyset$. Two (Boolean) ABoxes $\mathcal{A}$ and $\mathcal{A}'$ are *equivalent*, denoted by $\mathcal{A} \equiv \mathcal{A}'$, if $M(\mathcal{A}) = M(\mathcal{A}')$. An assertion α is *entailed* by a Boolean ABox $\mathcal{A}$, written as $\mathcal{A} \models \alpha$, if $M(\mathcal{A}) \subseteq M(\{\alpha\})$. Classical $\mathcal{ALCO}^{@}$-ABoxes can equivalently be compiled to Boolean $\mathcal{ALCO}$-ABoxes (and vice versa) — the translation in the first direction is exponential, in the other direction it is linear [4]. *Consistency checking* and *entailment* for classical ABoxes are standard inference problems and supported by all DL reasoners[1], while, to the best of our knowledge, no state of the art reasoner directly supports these inferences for Boolean ABoxes. Reasoning in $\mathcal{ALCO}^{+}$ is NExpTime complete [9]; for $\mathcal{ALCO}^{@}$ it is PSpace complete [10].

ABox Update An ABox can be used to represent knowledge about the state of some world. An *update* contains information on changes that have taken place in that world.

Definition 1 (Update). *An* update $\mathcal{U} = \{\delta(\bar{t})\}$ *contains a single literal, i.e.* $\delta(\bar{t})$ *is of the form* $A(a)$, $\neg A(a)$, $r(a, b)$, *or* $\neg r(a, b)$ *with* A *a concept name,* r *a role name, and* a, b *individual names.*[2]

Intuitively, an update literal $\delta(\bar{t})$ says that this literal holds after the change of the world state. The formal semantics of updates given in [4] defines, for every interpretation $\mathcal{I}$, a successor interpretation $\mathcal{I}^{\mathcal{U}}$ obtained by changing this model according to the update. Given an ABox $\mathcal{A}$, all its models are considered to be possible current states of the world. The goals is then to find an updated ABox $\mathcal{A} * \mathcal{U}$ that has exactly the successor of the models of $\mathcal{A}$ as its models, i.e., $\mathcal{A} * \mathcal{U}$

[1] A list of DL reasoners is available at `http://www.cs.man.ac.uk/~sattler/reasoners.html`.

[2] In [4], an update is defined as a consistent set of literals, not as a single literal. Updating an ABox $\mathcal{A}$ with a set of literals can in our setting be achieved by iteratively updating $\mathcal{A}$ with the individual literals.

must be such that $M(\mathcal{A} * \mathcal{U}) = \{\mathcal{I}^{\mathcal{U}} \mid \mathcal{I} \in M(\mathcal{A})\}$. In general, such an updated ABox need not exists.

$$
\begin{aligned}
(\exists r.C)^{\mathcal{U}} &= \Big(\bigsqcap_{a \in \mathsf{Obj}(\mathcal{U})} \neg\{a\} \sqcap \exists r.C^{\mathcal{U}}\Big) \sqcup \exists r.\Big(\bigsqcap_{a \in \mathsf{Obj}(\mathcal{U})} \neg\{a\} \sqcap C^{\mathcal{U}}\Big) \\
&\quad \sqcup \bigsqcup_{a,b \in \mathsf{Obj}(\mathcal{U}), r(a,b) \notin \mathcal{U}} (\{a\} \sqcap \exists r.(\{b\} \sqcap C^{\mathcal{U}})) \sqcup \bigsqcup_{\neg r(a,b) \in \mathcal{U}} (\{a\} \sqcap @_b C^{\mathcal{U}}) \\
(\forall r.C)^{\mathcal{U}} &= \Big(\bigsqcup_{a \in \mathsf{Obj}(\mathcal{U})} \{a\} \sqcup \forall r.C^{\mathcal{U}}\Big) \sqcap \forall r.\Big(\bigsqcup_{a \in \mathsf{Obj}(\mathcal{U})} \{a\} \sqcup C^{\mathcal{U}}\Big) \\
&\quad \sqcap \bigsqcap_{a,b \in \mathsf{Obj}(\mathcal{U}), r(a,b) \notin \mathcal{U}} (\neg\{a\} \sqcup \forall r.(\neg\{b\} \sqcup C^{\mathcal{U}})) \sqcap \bigsqcap_{\neg r(a,b) \in \mathcal{U}} (\neg\{a\} \sqcup @_b C^{\mathcal{U}})
\end{aligned}
$$

Fig. 1. Constructing $C^{\mathcal{U}}$ for $\mathcal{ALCO}^{@}$

The minimal DLs that contain both the basic DL $\mathcal{ALC}$ and are closed under ABox updates are $\mathcal{ALCO}^{@}$ and Boolean $\mathcal{ALCO}$. For $\mathcal{ALCO}^{@}$, updated ABoxes are exponential in the size of the original ABox and the update. The DL $\mathcal{ALCO}^{+}$ admits updated ABoxes that are exponential in the size of the update, but polynomial in the size of the original ABox. This is the reason why, in this work, we focus on $\mathcal{ALCO}^{+}$ and $\mathcal{ALCO}^{@}$. The following two propositions, which are simplified and streamlined versions of the ones given in [4], tell us how updated ABoxes can be computed for these two DLs:

Proposition 1 (Updated ABox for $\mathcal{ALCO}^{+}$). *Let $\alpha^{\mathcal{U}}$ be the concept (role) obtained by replacing every occurrence of*

- *A by $A \sqcap \neg\{a\}$ if $\mathcal{U} = \{A(a)\}$; and by $A \sqcup \{a\}$ if $\mathcal{U} = \{\neg A(a)\}$;*
- *r by $r \sqcap \neg\{(a,b)\}$ if $\mathcal{U} = \{r(a,b)\}$; and by $r \sqcup \{(a,b)\}$ if $\mathcal{U} = \{r(a,b)\}$.*

Let the ABox $\mathcal{A}'$ be defined as

$$\mathcal{A}' = \bigwedge(\mathcal{A} \cup \mathcal{U}) \vee \bigwedge(\mathcal{A}^{\mathcal{U}} \cup \mathcal{U}), \tag{1}$$

where the ABox $\mathcal{A}^{\mathcal{U}}$ is defined as $\mathcal{A}^{\mathcal{U}} = \{\alpha^{\mathcal{U}}(\bar{t}) \mid \alpha(\bar{t}) \in \mathcal{A}\}$. Then $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$.

Intuitively, there is one disjunct $(\mathcal{A} \cup \mathcal{U})$ for the case that the update already held before the update, and one disjunct $(\mathcal{A}^{\mathcal{U}} \cup \mathcal{U})$ for the case that its negation did.

The DL $\mathcal{ALCO}^{@}$ lacks role operators, and, hence, the construction of the updated quantifier concepts is complicated — it is depicted in Figure 1. Here $\mathsf{Obj}(\mathcal{U})$ denotes all the individuals that occur in the update $\mathcal{U}$. For concept names the construction is as in $\mathcal{ALCO}^{+}$.

Proposition 2 (Updated ABox for $\mathcal{ALCO}^{@}$). *For $\mathcal{ALCO}^{@}$ the ABox $\mathcal{A}^{\mathcal{U}}$ is defined as*

$$
\begin{aligned}
\mathcal{A}^{\mathcal{U}} = \{&C^{\mathcal{U}}(a) \mid C(a) \in \mathcal{A}\} \cup \{r(a,b) \mid r(a,b) \in \mathcal{A} \wedge \neg r(a,b) \notin \mathcal{U}\} \cup \\
&\{\neg r(a,b) \mid \neg r(a,b) \in \mathcal{A} \wedge r(a,b) \notin \mathcal{U}\}.
\end{aligned}
$$

Let $\mathcal{A}'$ be as defined in (1). Then $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$.

In the following, we want to illustrate the usefulness of ABox updates by a simple example. In this example, it is convenient to use also a TBox. TBoxes are a very useful feature of DLs that allow us to introduce abbreviations for complex concepts. A TBox $\mathcal{T}$ is a finite set of concept definitions of the form $A \equiv C$, where A is a concept name (called a defined concept) and C is a complex concept. This TBox is acyclic if it does not contains multiple or cyclic definitions. Acyclic TBoxes introduce abbreviations for complex concepts, but these abbreviations can be expanded out [5]. This makes it possible to work with ABox updates also in the presence of acyclic TBoxes as long as defined concept names do not occur in the update. This restriction avoids semantic problems [3] such as the ramification problem.

Example 1 (Medical Record – Acetylsalicylic Acid). The following concept definitions could be part of a bigger medical ontology for pain treatment. It states under what conditions a treatment with acetylsalicylic acid (ASA) is indicated for a patient, in terms of both anamnesis and diagnosis results, under the additional safety condition that there must not be a contraindication for "similar" patients:[3]

$$
\begin{aligned}
\text{ASA-indicated} \equiv\ & \text{ASA-tolerant} \sqcap \text{ASA-Diagnosis} \sqcap \\
& \forall \text{similar\_patient.ASA-tolerant} \\
\text{ASA-tolerant} \equiv\ & \neg\text{Pregnant} \sqcap \neg\text{Atopic} \sqcap \neg\text{Infant} \sqcap \neg\text{Child} \\
\text{ASA-Diagnosis} \equiv\ & (\text{Migraine} \sqcup \text{Tension\_Headache} \sqcup \text{Cluster\_Headache} \sqcup \\
& \quad \text{Drug-induced\_Headache} \sqcup \text{Impingement\_Syndrome} \sqcup \\
& \qquad \vdots \\
& \quad \text{HIV\_Peripheral\_Neuropathy}) \sqcap \\
& \neg\text{Bleeding\_Diathesis} \sqcap \neg\text{Heart\_Disease} \sqcap \\
& \neg\text{Renal\_Disease} \sqcap \neg\text{Peptic\_Ulcer} \\
\text{Migraine} \equiv\ & \ldots
\end{aligned}
$$

Assume that the ABox describing the medical record of the patient Mary, who has come to the hospital because she suffers from migraine, includes the ABox assertions in the first line below. In addition, assume that this ABox contains the information that the patient Jane is similar to Mary:

$$
\begin{gathered}
\text{ASA-Diagnosis} \sqcap \text{ASA-tolerant}(\text{Mary}), \forall \text{similar\_patient.ASA-tolerant}(\text{Mary}), \\
\text{similar\_patient}(\text{Mary}, \text{Jane}), \text{similar\_patient}(\text{Jane}, \text{Mary}).
\end{gathered}
$$

The DL reasoner can infer from this information that Mary belongs to the concept ASA-indicated, and that Jane belongs to the concept ASA-tolerant. But assume that, at her next visits, Mary tells the doctor that she is now pregnant. If her medical record is *updated* with {Pregnant(Mary)}, then we can conclude that an ASA treatment is no longer possible for Mary since the updated ABox implies ¬ASA-indicated(Mary). However, it also implies ¬ASA-indicated(Jane)

[3] We assume here that the (reflexive and symmetric) similarity relation between patients is computed by some separate, non-DL mechanism [11].

since there is now a patient similar to Jane (i.e., Mary) that is not ASA tolerant. To avoid this (obviously unintended) consequence, we must additionally update the ABox with $\neg$similar_patient(Mary, Jane) and $\neg$similar_patient(Jane, Mary) (unless we have learnt that Jane is now also pregnant).

3 Optimizations for ABox Updates

It turns out that a naive implementation of the update algorithms based on Proposition 1 or 2 is not practical. Even for very simple update problems — where simple means e.g. small initial ABoxes containing only literals — after only a few updates we obtain ABoxes so huge and redundant that the reasoners cannot handle them anymore. In this section we propose a range of techniques for obtaining less redundant updated ABoxes.

In particular we are looking for ABoxes that are smaller than, but equivalent to, the updated ABoxes. In principle this could be done by enumerating ever bigger ABoxes, and checking for equivalence to the updated ABox. This is not likely to be practical, though. Instead we focus on logical transformations for obtaining smaller updated ABoxes. Since these transformations can be computationally expensive themselves, we also identify fragments of the transformations that we expect to be relatively cheap. The proposed techniques are each motivated by avoidable redundancy that we observed in practical examples. We present the various techniques for obtaining smaller updated ABoxes individually; they can be combined in a modular fashion.

Updating Boolean ABoxes Updating an ABox according to Proposition 1 or 2 results in a Boolean ABox. In [4] this updated ABox is transformed to a non-Boolean ABox using the @-constructor, before it is updated again. The following observation shows that Boolean ABoxes can directly be updated again by updating the individual assertions, avoiding the transformation.

Observation 1 (Distributivity of Update) *Update distributes over the connectives conjunction and disjunction in Boolean ABoxes; i.e.*

$$(\mathcal{A}_1 \boxtimes \mathcal{A}_2) * \mathcal{U} \equiv (\mathcal{A}_1 * \mathcal{U}) \boxtimes (\mathcal{A}_2 * \mathcal{U}),$$

where $\boxtimes$ denotes either $\wedge$ or $\vee$ (negation can be pushed inside the assertions).

By updating a Boolean ABox directly we also obtain a slightly more compact representation than the original one — the update $\mathcal{U}$ is no longer contained in two disjuncts:

Observation 2 (Updating Boolean ABoxes) *For a Boolean ABox $\mathcal{A}$ (we assume negation has been pushed inside the assertions), let the updated ABox $\mathcal{A}'$ be defined as*

$$\mathcal{A}' = (\mathcal{A} \circledast \mathcal{U}) \wedge \bigwedge \mathcal{U}.$$

Here $\mathcal{A} \circledast \mathcal{U}$ is defined recursively as

$$\begin{array}{lcl} \alpha \circledast \mathcal{U} & = & \alpha \vee \alpha^{\mathcal{U}} \\ (\alpha \boxtimes \mathcal{B}) \circledast \mathcal{U} & = & (\alpha \circledast \mathcal{U}) \boxtimes (\mathcal{B} \circledast \mathcal{U}) \end{array}$$

where $\boxtimes$ denotes $\wedge$ or $\vee$, α is an assertion, and $\{\alpha\}^{\mathcal{D}}$ is defined as in Proposition 1 (or 2) for $\mathcal{ALCO}^{+}$ (or for $\mathcal{ALCO}^{@}$, respectively). Then $\mathcal{A} * \mathcal{U} \equiv \mathcal{A}'$.

Determinate Updates Looking at the construction of updated ABoxes, we see that from an ABox $\mathcal{A}$ by an update we get a disjunction $\mathcal{A} \vee \mathcal{A}^{\mathcal{U}}$. This causes a rapid growth of the updated ABox. If, however, either the update or its negation is entailed by the ABox $\mathcal{A}$, then one of the disjuncts is inconsistent and can be removed:

Observation 3 (Determinate Updates) *For Boolean ABox $\mathcal{A}$, update $\mathcal{U} = \{\delta\}$, and updated ABox $\mathcal{A}'$ we have that $\mathcal{A}' \equiv \mathcal{A}$ if $\mathcal{A} \models \delta$; and $\mathcal{A}' \equiv \mathcal{U} \cup \mathcal{A}^{\mathcal{U}}$ if $\mathcal{A} \models \neg\delta$.*[4] *Otherwise, if neither $\mathcal{A} \models \delta$ nor $\mathcal{A} \models \neg\delta$, both $\mathcal{A}^{\emptyset}$ and $\mathcal{A}^{\mathcal{U}}$ are consistent with $\mathcal{U}$.*

Detecting this type of situation requires up to two reasoning steps: $\mathcal{A} \models \delta$ and $\mathcal{A} \models \neg\delta$, resulting in a tradeoff between time and space efficiency.

Exploiting the Unique Name Assumption The common unique name assumption (UNA) means that no two individual names may denote the same object. The constructions from Proposition 1 and 2 do not take the UNA into account; but we can construct simpler updated ABoxes by keeping track of the individuals $\bar{s}$ and $\bar{t}$ that an assertion $\gamma(\bar{s})$ refers to when updating it with $\delta(\bar{t})$:

Example 2 (Exploiting UNA). If we update the ABox $\mathcal{A} = \{A(i)\}$ with $\mathcal{U} = \{\neg A(j)\}$, we can easily obtain $A(i)$, instead of $A \sqcup \{j\}(i)$ using the standard construction. But next consider the ABox $\mathcal{A} = \{\forall r.(\{j\} \sqcap A)(i)\}$, updated by $\mathcal{U} = \{A(k)\}$. As part of the update construction we obtain $\forall r.(\{j\} \sqcap (A \sqcap \neg\{k\}))(i)$ which can be simplified using UNA to $\forall r.(\{j\} \sqcap A)(i)$. Our implemented method for exploiting UNA cannot detect this latter case.

This UNA-based construction is not costly at all. It cannot identify all cases where the UNA admits a more concise updated ABox, though.

Omitting Subsuming Disjuncts and Entailed Assertions Intuitively, in a disjunction we can omit the "stronger" of two disjuncts: Let the disjunction $(\mathcal{A} \vee \mathcal{A}^{\mathcal{U}})$ be part of an updated ABox. If $\mathcal{A} \models \mathcal{A}^{\mathcal{U}}$ (or $\mathcal{A}^{\mathcal{U}} \models \mathcal{A}$) then $(\mathcal{A} \vee \mathcal{A}^{\mathcal{U}}) \equiv \mathcal{A}^{\mathcal{U}}$ (or $(\mathcal{A} \vee \mathcal{A}^{\mathcal{U}}) \equiv \mathcal{A}$). Detecting subsuming disjuncts in general requires reasoning. But by a simple, syntactic check we can detect beforehand some cases where one of the disjuncts $\mathcal{A}^{\mathcal{U}}$ and $\mathcal{A}$ will subsume the other. Then the computation

[4] The latter of these two observations is from [4].

of subsuming disjuncts can be avoided. We say that an occurrence of a concept or role name δ in an assertion is *positive*, if it is in the scope of an even number of negation signs, and *negative* otherwise; δ *occurs only positively (negatively)* in an assertion if every occurrence of δ is positive (negative).

Observation 4 (Detecting Subsuming Disjuncts) *If for an ABox $\mathcal{A}$, updated with update $\mathcal{U} = \{(\neg)\delta(\bar{t})\}$, we have that:*

(1) if the update is positive (i.e. $\delta(\bar{t})$) then
- *if δ occurs only positively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{U}} \models \mathcal{A}$; and*
- *if δ occurs only negatively in $\mathcal{A}$ then $\mathcal{A} \models \mathcal{A}^{\mathcal{U}}$.*

(2) if the update is negative (i.e. $\neg\delta(\bar{t})$) then
- *if δ occurs only positively in $\mathcal{A}$ then $\mathcal{A} \models \mathcal{A}^{\mathcal{U}}$; and*
- *if δ occurs only negatively in $\mathcal{A}$ then $\mathcal{A}^{\mathcal{U}} \models \mathcal{A}$.*

Conversely, we can also avoid updating entailed assertions: Let $\mathcal{A}$ be an ABox and $\mathcal{U}$ an update. If $\mathcal{U} \models \alpha$ or $\mathcal{A} \setminus \{\alpha\} \models \alpha$ for some assertion $\alpha \in \mathcal{A}$, then $\mathcal{A} * \mathcal{U} \equiv (\mathcal{A} \setminus \{\alpha\}) * \mathcal{U}$. Removing all entailed assertions might be too expensive in practice; one might try doing this periodically.

Propositional ABoxes Sometimes we do not need the full power of DL reasoning, but propositional reasoning is enough: We call a Boolean ABox *propositional* if it does not contain quantifiers. For propositional ABoxes we could in principle use progression algorithms for propositional logic [12] and efficient SAT-technology, since an updated propositional ABox is propositional, too.

Independent Assertions Next we address the question under which conditions an assertion in an ABox is not affected by an update.We say that assertion α in an ABox $\mathcal{A}$ is independent from update $\mathcal{U} = \{\delta\}$ iff $\mathcal{A} * \mathcal{U} \equiv \alpha \wedge (\mathcal{B} * \mathcal{U})$ where $\mathcal{B} = \mathcal{A} \setminus \{\alpha\}$. The more independent assertions we can identify, the more compact our ABox representation becomes.

Detecting this in all cases requires reasoning steps and thus is costly. It is easy, though, to syntactically detect some of the independent assertions:

Observation 5 (Independent Assertion) *For an ABox $\mathcal{A}$ in negation normal form and update $\mathcal{U} = \{(\neg)\delta(\bar{t}_1)\}$, the assertion $\alpha(\bar{t}_2) \in \mathcal{A}$ is independent if $\delta \notin \mathsf{sub}(\alpha)$. It is also independent if $\mathcal{A} \models \bar{t}_1 \neq \bar{t}_2$, δ occurs in α only outside the scope of a quantifier, and for all subconcepts $@_i C$ of α the assertion $C(i)$ is independent of $\mathcal{U}$.*

4 Reasoning with Boolean ABoxes

As we have seen in the previous sections, updated ABoxes are Boolean $\mathcal{ALCO}^{@}$- or $\mathcal{ALCO}^{+}$-ABoxes, so that an intelligent agent built on top of ABox update needs Boolean ABox reasoning. Reasoning with $\mathcal{ALC}$-LTL formulas [13] requires Boolean ABox reasoning, too. However, Boolean ABox reasoning is not directly supported by DL reasoners. In this section, we present four different reasoning methods that can handle Boolean ABoxes:

- one where a DL reasoner operates on single disjuncts of an ABox in DNF;
- one which uses Otter, a first-order theorem prover;
- one which uses a consistency preserving reduction from a Boolean ABox to a non-Boolean ABox; and
- one which is based on propositional satisfiability testing modulo theories — the DPLL(T) approach.

Replacing every assertion in a Boolean ABox $\mathcal{A}$ with a propositional letter results in a propositional formula $F_{\mathcal{A}}$. The ABox $\mathcal{A}$ is a *Boolean ABox in CNF (resp. DNF)* if $F_{\mathcal{A}}$ is in CNF (resp. DNF). The first approach works on Boolean ABoxes in DNF while the other approaches are based on CNF.

We do not use the equivalence-preserving, exponential transformation from [4] for compiling the @ constructor away. Instead we simulate the @-operator by a universal role [14]; this consistency-preserving transformation is linear.

We use Pellet as a DL reasoner because it supports nominals, query-answering and pinpointing [15].

The DNF Approach A Boolean ABox in DNF is consistent iff it contains a consistent disjunct. We can employ a DL reasoner to decide the consistency of each disjunct. We refer to this approach as Pellet-DNF. A drawback of this approach is that we will see that the less redundant updated ABoxes are in CNF, and thus require a costly translation to DNF (using de Morgan's laws).

The Theorem Prover Approach The DL $\mathcal{ALCO}^+$ admits smaller updated ABoxes than $\mathcal{ALCO}^@$ [4]; however, its role operators are not supported by current mature DL reasoners. By translating $\mathcal{ALCO}^+$ to first order logic [16] we can use theorem provers that can cope with Boolean role constructors. We chose to use Otter [17] because it supports query-answering via answer literals [18]; this is useful e.g. for parametric actions, which are to be instantiated to concrete actions. After a few experiments we chose to configure Otter to use hyperresolution combined with Knuth-Bendix-rewriting, plus the set-of-support strategy.

The Reduction Approach We can linearly compile Boolean $\mathcal{ALCO}^@$-ABoxes to classical $\mathcal{ALCO}^@$-ABoxes [4]. Then, simulating the @-operator by a universal role, we can directly use a standard DL reasoner; this approach is henceforth called Pellet-UR.

The DPLL(T) Approach Most modern SAT-solvers [19, 20] are variants of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [21, 22]. Such a SAT-solver exhaustively applies transition rules[5] to generate and extend a partial interpretation and thus decides satisfiability of a propositional formula in CNF. One of the strengths of the DPLL procedures is that they can efficiently prune the search space by building and learning backjump clauses [24].

[5] See [23] for the details.

The DPLL(T) approach combines a DPLL procedure with a theory solver that can handle conjunctions of literals in the theory to solve the satisfiability problem modulo theories (SMT) [23]. In DPLL(T) a DPLL procedure works on the propositional formula obtained by replacing the theory atoms with propositional letters. Whenever the DPLL procedure extends the current partial interpretation by a new element the theory solver is invoked to check consistency of the conjunction of the theory atoms corresponding to the partial, propositional interpretation. If the theory solver reports an inconsistency, the DPLL procedure will backjump and thus the search space is pruned.

The consistency problem of Boolean ABoxes can be viewed as an instance of SMT where ABox assertions are the theory atoms and a DL reasoner serves as theory solver.

The non-standard DL inference of pinpointing [25, 26] is highly relevant to this approach. Explaining why an ABox is inconsistent is an instance of the pinpointing problem, where an explanation is a minimal sub-conjunction of the input ABox, containing only those assertions that are responsible for the inconsistency. Based on these explanations in the DPLL(T) approach we can build better backjump clauses [23].

We implemented an algorithm based on the DPLL(T) approach with the strategy of MiniSat [19]. Pellet was chosen as the theory solver because it supports pinpointing. Henceforth we call this approach Pellet-DPLL.

Propositional Reasoning For the case where we can identify propositional ABoxes we have developed and implemented a simple, specialized method. Reasoning there is reduced to efficient list operations. This reasoner is used to supplement the other reasoning approaches (if possible).

5 Experimental Results

In this section, we evaluate the efficiency of the different update and reasoning mechanisms. The relevant measures are the time needed for computing the updated ABox together with its size, and the efficiency of reasoning with it.

An update algorithm based on Proposition 1 or 2 generates Boolean ABoxes in DNF, while an algorithm based on Proposition 2 outputs ABoxes in CNF. Of course, every Boolean ABox can equivalently be represented in CNF or in DNF; however, this transformation (using De Morgan's laws) is rather expensive. The performance of reasoning with updated ABoxes strongly depends on the choice of underlying representation. We use several types of testing data:

- we use a set of randomly generated Boolean ABoxes in CNF;
- we use a set of random ABoxes, Updates, and Queries; and
- we use the Wumpus world [27].

We distinguish two main types of update algorithms that we implemented:

- In one we compute updated ABoxes in DNF; we call this the DNF approach.

– Or we compute updated ABoxes in CNF; we call this the CNF approach.

Both approaches are further parametrized by using different reasoners, and a different combination of optimization techniques. We have implemented the different ABox update algorithms in ECLiPSe-Prolog.

The reasoning methods have already been described in Section 4. We call a reasoning method *hybrid* if it resorts to our propositional reasoner whenever possible; for example, we then speak of hybrid Pellet-UR.

5.1 Representation: DNF or CNF?

We have used both the Wumpus world and the random update examples to compare DNF and CNF based update algorithms (with and without optimizations). CNF representation consistently proved to be superior: The DNF approach quickly drowns in redundant information. This is because to compute an updated ABox in DNF is to include both the update and all the non-affected information in both disjuncts. Detecting subsuming disjuncts and determinate updates alleviates this problem, but does not eliminate it. By avoiding this redundancy we immediately obtain an updated ABox in CNF. On DNF-based updated ABoxes Pellet-DNF performs best — the other methods suffer from the expensive conversion to CNF. In the following we only consider the CNF-based representation of updated ABoxes.

5.2 Consistency Checking for Boolean ABoxes in CNF

We implemented a random generator of Boolean $\mathcal{ALC}$-ABoxes, which randomly generates a propositional formula in CNF and then assigns a randomly generated assertion to each propositional letter. Several parameters are used to control the shape of the generated Boolean ABoxes (the numbers in parentheses indicate the upper bound on the parameters we used): the number n_1 of literals in a clause (53), the number n_2 of propositional letters (36), the number n_3 of clauses (83), the number d of nested roles in a concept assertion (23), the number ncs of the constructors in a concept assertion (106), the numbers nc, nr, and ni of concept names, role names, and individual names in an assertion (12 each), and the probability pr of generating a role assertion (0.2).

In Figure 2, we plot the runtimes of Pellet-DPLL and Pellet-UR on these testing data against the number of symbols in the Boolean ABox. The points plotted as + indicate the runtime of Pellet-DPLL while those plotted as × indicate the runtime of Pellet-UR. We depict the performance on consistent and inconsistent Boolean ABoxes separately — the testing data contained more consistent than inconsistent Boolean ABoxes.

We can see that the runtime of Pellet-UR linearly increases with the size of the input (the bar from the lower left to the upper right corner). On inconsistent ABoxes Pellet-DPLL also exhibits a linear increase in runtime, while on consistent ABoxes the runtime is less predictable. Pellet-DPLL performs better on all of the inconsistent Boolean ABoxes. On most of the consistent ABoxes,

the Pellet-UR approach does better. This is due to the fact that in Pellet-DPLL the frequent invocations of the theory solver Pellet are more likely to pay off if inconsistency of the current, partial model can be detected often: We then can build a back-jump clause that helps to prune the search space. The runtimes of Pellet-UR are about the same on both consistent and inconsistent input data.

For Otter the conversion from ABoxes in CNF to full first order CNF proved to be a big obstacle, as did the conversion to DNF for Pellet-DNF.

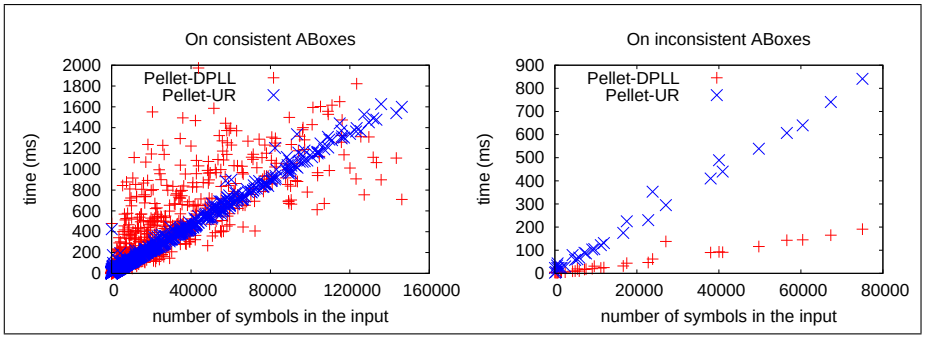

Fig. 2. Experimental Results for Pellet-DPLL and Pellet-UR

5.3 Random Updates

We have extensively experimented with a set of randomly generated ABoxes and updates. Initial ABoxes were between two and thirty assertions in size.We were mostly interested in runtime and space consumption for iterated updates. We could make a number of interesting observations:

- The cheap UNA-based concept update construction always paid.
- The reasoning needed to identify determinate updates pays in the long run.
- Syntactically detecting subsuming disjuncts worked, too. Doing so using a reasoner proved to be too expensive.
- Identifying all entailed assertions to shrink the ABoxes proved to be too expensive, too.
- Resorting to our dedicated propositional reasoner whenever possible resulted in significantly better performance.
- We can keep updated ABoxes much smaller at a low cost by syntactically identifying independent assertions.

Updating an ABox according to [4] is a purely syntactic procedure. But if we iteratively update ABoxes, then in the long run we get both a lower space and time consumption by calling a reasoner to identify determinate updates. Using our propositional reasoner whenever possible for this resulted in better performance. If identifying determinate updates required DL reasoning then Pellet-UR

performed slightly better than Pellet-DPLL. This is due to the fact that less updates were determinate than not, and thus inconsistency was not detected often. On a subset of the random examples where there were more determinate updates Pellet-DPLL performed better than Pellet-UR. The runtimes for Otter widely varied: converting CNF-ABoxes to full first order CNF proved the bottleneck. Pellet-DNF was not competitive because of the expensive conversion to DNF.

We could also identify characteristics of initial ABoxes that allow to predict performance: If the initial ABox does not contain nested quantifiers then performance is acceptable; e.g. we can iteratively apply 300 singleton updates to a fifteen assertion ABox in 90 seconds, without a significant increase in size. If the initial ABox contains nested quantifiers space consumption quickly grows out of bounds. This is because we then cannot cheaply identify independent assertions and use the UNA-based concept update construction. For nested quantifiers using $\mathcal{ALCO}^+$ instead of $\mathcal{ALCO}^@$ helps to reduce space consumption; but this still does not result in satisfactory overall performance.

5.4 The Wumpus World

The Wumpus World [27] is a well-known challenge problem in the reasoning about action community. It consists of a grid-like world: cells may contain pits, one cell contains gold, and one the fearsome Wumpus. The agent dies if she enters a cell containing a pit or the Wumpus. But she carries one arrow so that she can shoot the Wumpus from an adjacent cell. If the agent is next to a cell containing a pit (Wumpus), she can detect that one of the surrounding cells contains a pit (the Wumpus), but doesn't know which one. She knows the contents of the already visited cells. Getting the gold without getting killed is the agent's goal.

At each step, the agent performs sensing to learn whether one of the adjacent cells contains a pit or the Wumpus. Since the sensing results are disjunctive, we cannot treat them via ABox updates. But the properties sensed are static (i.e., cannot change once we know them): We can simply adjoin the sensing results to the ABox serving as the agent's current world model. The effects of the agent's (non-sensing) actions (like moving to another cell) are modelled as ABox update.

The Wumpus World can be modelled in different ways. In the simplest model, the initial ABox contains the connections between the cells, the agent's location, and the facts that the agent carries an arrow, and that the Wumpus is alive (Model PL1). For this, Boolean combinations of concept/role literals are enough. In Model PL2, we include the fact that the Wumpus is at exactly one location by enumerating all possible cases in a big disjunction. We turn PL1 into a DL problem by including the information $\exists \mathrm{at}.\top(\mathrm{wumpus})$ (Model DL1). Model DL2 is obtained from PL2 by adding this same assertion, which here is redundant. Table 2 shows the runtimes, where n/a stands for unavailable expressivity and * for non-termination in 15 minutes. For the propositional models we also used the action programming language Flux [2].

Pellet-DNF, and to a lesser extent also Otter, again had difficulties with the necessary input conversion. Pellet-UR proved to be the best DL reasoner in this setting. This is due to the fact that this domain requires query-answering: The

Model	Prop	hybrid Otter	hybrid Pellet-UR	Flux
8x8 PL1	0.26 s	0.26 s	0.26 s	14.9 s
8x8 PL2	16.9 s	16.9 s	16.9 s	n/a
4x4 DL1	n/a	36.4 s	5.5 s	n/a
4x4 DL2	n/a	*	23.93 s	n/a

Table 2. Runtimes for the Wumpus World.

agent e.g. needs to know for which values of x and y we have that $\text{at}(\text{agent}, x) \wedge \text{connected}(x, y)$. Pellet-DPLL is the only reasoner that lacks direct support for query-answering. Thus, for query $C(x)$, we check for every individual name $i \in \mathsf{N_I}$ whether $C(i)$ holds — this results in bad performance for Pellet-DPLL.

The propositional reasoner performs quite well on the propositional models. Including more information wrt. the Wumpus' location results in worse performance. We used Model DL2 to see if it pays to identify all entailed assertions: after omitting the entailed $\exists \text{at}.\top(\text{wumpus})$ the model is propositional again. In practice this proved too costly. The other observations from Section 5.3 also hold in this domain. Removing assertions entailed by the update sometimes did help, though: Once the Wumpus is found, the assertion $\exists \text{at}.\top(\text{wumpus})$ is entailed by the respective update and we can then resort to efficient propositional reasoning.

6 Summary and Future Work

In this work, we have investigated implementation techniques for ABox update, and for reasoning with (updated) Boolean ABoxes. We have introduced and evaluated several optimizations of the ABox update algorithms in [4]. The lessons learnt were: Using CNF-representation of updated ABoxes is strongly recommended. The (incomplete) syntactic techniques for exploiting the unique name assumption, and detecting subsuming disjuncts and independent assertions have also resulted in an improved performance. The benefit of identifying determinate updates made up for the associated reasoning costs. Other techniques requiring DL reasoning in general proved to be too expensive; but removing some entailed assertions helped in the Wumpus world.

Regarding the investigated reasoning methods for Boolean ABoxes, we have come to the following conclusions. Pellet-DNF is the best reasoner for Boolean ABoxes in DNF. For consistency checking of ABoxes in CNF, Pellet-DPLL and Pellet-UR worked best. Pellet-DPLL did better for detecting an actual inconsistency, while it performed worse than Pellet-UR on most of the consistent Boolean ABoxes. On the randomly generated update examples, Pellet-UR also performed slightly better than Pellet-DPLL because inconsistency was not detected often. On a subset where the updates were mostly determinate, Pellet-DPLL outperformed Pellet-UR. If query-answering is among the reasoning tasks, then Pellet-UR is to be preferred over Pellet-DPLL because of Pellet's direct support for this inference.

It would be interesting to develop heuristics for finding suitable individual names as well as other optimizations for query-answering in the DPLL(T) approach. The performance of the DPLL(T) approach also depends on the performance of the SAT solver and the pinpointing service. Thus Pellet-DPLL can benefit from more efficient implementation of these tasks as well.

The tests on the Wumpus world confirmed that resorting to our dedicated propositional reasoner whenever possible is useful. In the Wumpus world, removing entailed assertions helped a lot. In contrast, for the randomly generated update examples, finding entailed assertions did not pay off.

Using Otter as a theorem prover might be considered somewhat unfair (to the theorem proving approach), since it is no longer actively maintained and optimized. The conversion to full first order CNF proved to be the biggest obstacle for Otter. We chose to use Otter because it supports query-answering, which is not supported by most current provers [28], but vital in some domains. If this is to change,[6] we can try to resort to state-of-the art theorem provers for reasoning in $\mathcal{ALCO}^+$. This may allow us to really exploit the fact that $\mathcal{ALCO}^+$ admits smaller updated ABoxes than $\mathcal{ALCO}^@$. Alternatively, one could also try to use a more dedicated reasoning system for $\mathcal{ALCO}^+$ [29].

Acknowledgments: Many thanks to Albert Oliveras for his help regarding the construction of a backjump clause in the DPLL(T) approach.

References

1. Levesque, H., Reiter, R., Lespérance, Y., Lin, F., Scherl, R.: GOLOG: A logic programming language for dynamic domains. Journal of Logic Programming (1997)
2. Thielscher, M.: FLUX: A Logic Programming Method for Reasoning Agents. Theory and Practice of Logic Programming (2005)
3. Baader, F., Lutz, C., Milicic, M., Sattler, U., Wolter, F.: Integrating Description Logics and Action Formalisms: First Results. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI 2005), AAAI Press (2005)
4. Liu, H., Lutz, C., Milicic, M., Wolter, F.: Updating Description Logic ABoxes. In: Proceedings of the Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR'06), AAAI Press (2006)
5. Baader, F., Calvanese, D., Mcguinness, D.L., Nardi, D., Patel-Schneider, P.F., eds.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
6. Drescher, C., Thielscher, M.: Integrating Action Calculi and Description Logics. In: Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI 2007). (2007)
7. Drescher, C., Liu, H., etal.: Putting abox updates into action. LTCS-Report 09-01, Dresden University of Technology, Germany (2009) See http://lat.inf.tu-dresden.de/research/reports.html.
8. Areces, C., de Rijke, M.: From Description Logics to Hybrid Logics, and Back. In: Advances in Modal Logic. (2001)
9. Tobies, S.: Complexity Results and Practical Algorithms for Logics in Knowledge Representation. PhD thesis, RWTH-Aachen, Germany (2001)

[6] cf. `www.cs.miami.edu/~tptp/TPTP/Proposals/AnswerExtraction.html`.

10. Areces, Blackburn, Marx: A road-map on complexity for hybrid logics. In: CSL: 13th Workshop on Computer Science Logic, LNCS, Springer-Verlag (1999)
11. Zezula, P., Batko, M., Dohnal, V., , Amato, G.: Similarity Search: The Metric Space Approach. Springer (2006)
12. Amir, E., Russell, S.J.: Logical Filtering. In: IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann (2003)
13. Baader, F., Ghilardi, S., Lutz, C.: LTL over Description Logic Axioms. In: Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR2008). (2008)
14. Bong, Y.: Description Logic ABox Updates Revisited. Master thesis, TU Dresden, Germany (2007)
15. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics (2007)
16. Borgida, A.: On the Relative Expressiveness of Description Logics and Predicate Logics. Artificial Intelligence (1996)
17. McCune, W.: OTTER 3.3 Manual. Computing Research Repository (2003)
18. Green, C.: Theorem Proving by Resolution as a Basis for Question-answering Systems. Machine Intelligence (1969)
19. Een, N., Sörensson, N.: An Extensible SAT-solver. In: International Conference on Theory and Applications of Satisfiability Testing (SAT). (2003)
20. de Moura, L., Bjørner, N.: Z3: An efficient SMT Solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Springer (2008)
21. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM (1960)
22. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-proving. Communications of the ACM (1962)
23. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in Satisfiability Modulo Theories. In: 18th International Conference on Term Rewriting and Applications, Springer (2007)
24. Zhang, L., Madigan, C.F., Moskewicz, M.H., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: International Conference on Computer-Aided Design (ICCAD'01). (2001)
25. Schlobach, S.: Non-Standard Reasoning Services for the Debugging of Description Logic Terminologies. In: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, (IJCAI-03), Morgan Kaufmann (2003)
26. Baader, F., Peñaloza, R.: Automata-Based Axiom Pinpointing. In: Proceedings of the 4th International Joint Conference on Automated Reasoning, (IJCAR 2008), Springer (2008)
27. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (2003)
28. Waldinger, R.J.: Whatever happened to deductive question answering? In: Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, (LPAR 07), Yerevan, Armenia, Springer (2007)
29. Schmidt, R.A., Tishkovsky, D.: Using tableau to decide expressive description logics with role negation. In: Proceedings of the 6th International Semantic Web Conference, ISWC 2007, Springer (2007)