Int. J. Appl. Math. Comput. Sci., *2011*, *Vol. 21*, *No. 1*, *83–95*
DOI: 10.2478/v10006-011-0006-1

EXPERIMENTAL ANALYSIS OF SOME COMPUTATION RULES IN A SIMPLE PARALLEL REASONING SYSTEM FOR THE $\mathcal{ALC}$ DESCRIPTION LOGIC

ADAM MEISSNER

Institute of Control and Information Engineering
Poznań University of Technology, pl. M. Skłodowskiej-Curie 5, 60–965 Poznań, Poland
e-mail: Adam.Meissner@put.poznan.pl

A computation rule determines the order of selecting premises during an inference process. In this paper we empirically analyse three particular computation rules in a tableau-based, parallel reasoning system for the $\mathcal{ALC}$ description logic, which is built in the relational programming model in the Oz language. The system is constructed in the lean deduction style, namely, it has the form of a small program containing only basic mechanisms, which assure soundness and completeness of reasoning. In consequence, the system can act as a convenient test-bed for comparing various inference algorithms and their elements. We take advantage of this property and evaluate the studied methods of selecting premises with regard to their efficiency and speedup, which can be obtained by parallel processing.

Keywords: parallel reasoning, lean deduction, $\mathcal{ALC}$ description logic, Oz language.

1. Introduction

Description logics (DLs) (Baader *et al.*, 2003) is the name of a family of formal systems mainly used for representing and processing terminological knowledge. A common criterion of classification for DLs is their language. In particular, special interest is given to a class of DLs whose core language is the $\mathcal{ALC}$ language ($\mathcal{ALC}$ DLs). Logics from this class have a reasonable expressivity and are, in many cases, decidable. Hence, they have been successfully applied in various domains, such as software engineering (Devanbu and Jones, 1997), object databases (Calvanese *et al.*, 1999), control in manufacturing (Rychtyckyj, 1996), action planning in robotics (De Giacomo *et al.*, 1996), medical expert systems (Rector *et al.*, 1998) and also the Semantic Web (*Semantic Web*, 2001). DLs generally provide the semantics of knowledge bases for systems constructed in the areas mentioned. In the case of the Semantic Web, $\mathcal{ALC}$ DLs stand behind the OWL-DL language (*OWL Web Ontology Language Overview*, 2004), which is widely used to represent the meaning of documents available in the WWW network.

All essential inference problems for $\mathcal{ALC}$ DLs are at least of PSPACE complexity. Thus, one of the biggest research challenges in this area is the development of tractable reasoning methods and systems. A reasoning system (also called a reasoner or an inference system) generally consists of a declarative part and a computational (execution) strategy. The declarative part encompasses the implementation of inference rules, while the execution strategy defines the way the rules are applied. A number of reasoning systems are implemented as logic programs in the Prolog language. This mainly follows from the fact that a program in Prolog is formally a set of logic formulas, that is to say, normal clauses. Therefore, many elements of the language and its computational model (e.g., operations on terms, the inference rule) can be successfully absorbed to the constructed system. This leads to the idea of *lean deduction* (Beckert and Possega, 1995), which consists in implementing inference systems as small programs equipped only with basic mechanisms necessary for soundness and completeness of the reasoning process. It is definitely not a way to construct highly efficient reasoners using sophisticated techniques for solving difficult problems, as, for example, FaCT++ (Tsarkov and Horrocks, 2006), Racer Pro (Wessel and Möller, 2005) or KAON2 (Hustadt *et al.*, 2004) in the domain of DLs.

However, this approach has many advantages. Lean reasoners are small, and hence not hard to verify. In contrast to complex systems, they can be easily modified and adapted to particular applications (Amir and Maynard-Zhang, 2004). Also, lean reasoners are surprisingly efficient for solving simple and less difficult problems because of lower overhead for handling internal components than in the case of sophisticated systems. Moreover, they can act as convenient test-beds for comparing various inference techniques, where absolute execution efficiencies are not as important as relative ones.

In this work we summarize and expand earlier results concerning a lean reasoning system for the $\mathcal{ALC}$ DL presented by Meissner (2009a; 2009b). Our approach extends the idea of lean deduction by parallel processing. Furthermore, the system is built in the *relational model* in the Oz language (Van Roy and Haridi, 2004). The model corresponds to logic programming (especially to Prolog) with regard to its declarative semantics. However, the operational semantics of relational programs, unlike in Prolog, are not fixed in the runtime environment but implemented as a *search engine*—a special object which executes a program. This makes it possible to run a program in various ways, particularly in parallel on distributed machines.

We take advantage of this property and construct a small reasoning procedure consisting of c.a. 50 lines of code (Meissner, 2009b), mainly for implementing inference rules of the tableau calculus for the $\mathcal{ALC}$ DL. It is one to two orders of magnitude shorter than sophisticated reasoners for description logics (Aslani and Haarslev, 2008; Liebig and Müller, 2007). In order to run the procedure, we use the parallel search engine (Schulte, 2000) available in the Mozart system (*The Mozart Programming System*, 2008), which is a programming platform for the Oz language. Experiments show a reasonable speedup obtained with the increasing number of processors added to the computational environment.

Meissner (2009a) considers a modified version of the system with a particular computation rule originating from the inference algorithm given by Schmidt-Schauß and Smolka (1991). It "is generally viewed as a sensible way of organising the expansion and the flow of control" within a sequential reasoning system for DLs (Baader *et al.*, 2003). We analyse and experimentally evaluate this method with regard to parallel processing, pointing out its advantages and drawbacks. In this paper, we additionally define another computation rule in order to overcome some disadvantages of the former approach. We test all the rules for the efficiency and for the speedup that can be obtained by parallelizing the computations.

The organisation of the paper is as follows. Section 2 contains the principles of tableau calculus for the $\mathcal{ALC}$ DL. In Section 3 we characterise the key elements of the inference algorithm, focusing particularly on the computation rules mentioned above. Section 4 specifies how tableau-based reasoning for the $\mathcal{ALC}$ DL is represented in the relational programming model. Some implementation details of the reasoning procedure are described in Section 5. Section 6 presents and discusses the results of experiments aimed at the comparison of the computation rules. Section 7 concludes the paper with some final remarks.

2. Tableau calculus for the $\mathcal{ALC}$ DL

First, we outline the syntax and the semantics of the $\mathcal{ALC}$ language. Then, we present one of basic inference problems for the $\mathcal{ALC}$ DL, that is to say, testing for the concept of satisfiability. Finally, we describe the classical tableau-based calculus in which the problem can be solved.

The $\mathcal{ALC}$ DL language contains two types of elementary expressions, i.e., *atomic descriptions* (or, synonymously, *names*) of *concepts* and atomic descriptions of *roles*. A concept is a set of individuals, called *instances* of this concept. A role is a binary relation holding between individuals. Any element of a role is called an *instance* of this role. Concepts, besides names, can also be represented by complex descriptions, which are built from simpler descriptions and special symbols called *concept constructors*. We use the letter A to denote a concept name and the letters C or D as symbols of any concept descriptions; the letter R stands for a role description. All these symbols can possibly be subscripted. The set of $\mathcal{ALC}$ DL concept constructors comprises five elements, namely, *negation* ($\neg C$), *intersection* ($C \sqcap D$), *union* ($C \sqcup D$), *existential quantification* ($\exists R.C$) and *value restrictions* ($\forall R.C$); expressions written in the parentheses are schemes of relevant concept descriptions. If it does not lead to a misunderstanding, in the sequel we use constructor names to call the descriptions created with them. For example, the expression of the form $C \sqcup D$ is called a *union*. We also often identify descriptions with their meaning (e.g., we say "a concept" instead of "a concept description"). Expressions of the form $C(x)$ and $R(x, y)$ are called *concept assertions* and *role assertions*, respectively. An expression of the first type states that the individual x is an instance of the concept C, while the latter expression declares that the pair of individuals $\langle x, y\rangle$ is an instance of the role R. The individual y is called a *filler* of the role R for x. Furthermore, when an assertion is built from an atomic concept, then it is called an *atomic assertion*.

In order to define the semantics of concept and role descriptions, we use an interpretation $\mathcal{I}$, which consists of the interpretation domain $\Delta^{\mathcal{I}}$ and the interpretation function $(\cdot)^{\mathcal{I}}$. The interpretation function assigns a subset of $\Delta^{\mathcal{I}}$ to every concept name and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every role description. The semantics of complex concepts are given as follows:

$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$$

$$(C \sqcap D)^{\mathcal{I}} = D^{\mathcal{I}} \cap C^{\mathcal{I}},$$
$$(C \sqcup D)^{\mathcal{I}} = D^{\mathcal{I}} \cup C^{\mathcal{I}},$$
$$(\exists R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid (\exists y)\, \langle x, y\rangle \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\},$$
$$(\forall R.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} \mid (\forall y)\, \langle x, y\rangle \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}.$$

Furthermore, there are two special concept descriptions, namely, $\top$ (top) and $\bot$ ($bottom$). The first one denotes the most general concept, that is, $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$, while the second represents the empty concept, i.e., $\bot^{\mathcal{I}} = \emptyset$. We say that the interpretation $\mathcal{I}$ *satisfies* the description C if it assigns a nonempty set to it. Such an interpretation is called a *model* of the concept C. The concept is *satisfiable* if there exists a model of it, otherwise it is *unsatisfiable*.

The (un)satisfiability of the concept C_0 can be checked by the classical tableau calculus (Baader *et al.*, 2003), which is sketched below. We assume that the concept C_0 (called an *input concept*) is initially converted to *negation normal form* (NNF), where the negation symbol appears directly before atomic concepts. The *tableau* for the input concept C_0 is the tree T, whose every node is labelled by a set containing, in general, concept and role assertions. For the sake of brevity we often say “a formula in the node” instead of “a formula in the label of the node”. The label of the root of T is one-element set $\{C_0(x_0)\}$, where x_0 is an arbitrarily given individual. Any other node (symbolized as $\mathcal{A}'$ or $\mathcal{A}''$) can be obtained from its direct ancestor $\mathcal{A}$ by applying one of the following *expansion rules*:

$\sqcap$-rule: if (a) $(C_1 \sqcap C_2)(x) \in \mathcal{A}$ and
(b) $\{C_1(x), C_2(x)\} \not\subseteq \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{C_1(x), C_2(x)\}$

$\sqcup$-rule: if (a) $(C_1 \sqcup C_2)(x) \in \mathcal{A}$ and
(b) $C_1(x) \notin \mathcal{A}$ and $C_2(x) \notin \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{C_1(x)\}, \mathcal{A}'' = \mathcal{A} \cup \{C_2(x)\}$

$\exists$-rule: if (a) $(\exists R.C)(x) \in \mathcal{A}$ and
(b) there is no y that $\{R(x, y), C(y)\} \subseteq \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{R(x, z), C(z)\}$ and z doesn't occur in $\mathcal{A}$

$\forall$-rule: if (a) $(\forall R.C)(x) \in \mathcal{A}$ and
(b) $R(x, y) \in \mathcal{A}$ and $C(y) \notin \mathcal{A}$
then $\mathcal{A}' = \mathcal{A} \cup \{C(y)\}$.

We say that a rule and a concept assertion $D(x)$ are *relevant* to each other if $D(x)$ matches the assertion occurring in the condition (a) of this rule. The application of the rule, however, may be *blocked* (in short: the rule is blocked) if the conditions (b) are not satisfied. A node of the tableau T is dangling if no expansion rule can be applied to it or if it contains a contradiction called a *clash*. In the latter case, the node is called a *clash node*, otherwise it is *clash free*. The clash can be detected by the following *clash rules*:

$clash1$-rule: if $\bot(x) \in \mathcal{A}$ or $(\neg\top)(x) \in \mathcal{A}$
then mark $\mathcal{A}$ as a clash node

$clash2$-rule: if $A(x) \in \mathcal{A}$ and $(\neg A)(x) \in \mathcal{A}$
then mark $\mathcal{A}$ as a clash node

The $clash2$-rule recognizes the presence of *complementary* assertions (i.e., $A(x)$ and $(\neg A)(x)$) in the node. It should be remarked that the application of this rule is restricted to *literals*, namely, to atomic concepts (*positive* literals) and their negations (*negative* literals), since the input concept is in NNF. Expansion rules and clash rules are both called *inference rules* or *tableau rules*.

A branch of the tableau is *closed* if it is ended by a clash node; otherwise, the branch is *open*. A tableau containing only closed branches is called a *closed tableau*. A concept C is *unsatisfiable* if and only if one can construct a closed tableau for it. Otherwise the concept is *satisfiable* and every fully expanded node at the end of an open branch is a straightforward representation of a model of the concept C.

In Fig. 1 we present a tableau constructed for the input concept resulting from the following example. Let us assume that we want to check if the concept $(\exists \mathit{HasChild}.\mathit{Good}) \sqcap (\exists \mathit{HasChild}.\mathit{Wise})$ is included in the concept $\exists \mathit{HasChild}.(\mathit{Good} \sqcap \mathit{Wise})$. Intuitively, the former denotes the set of all individuals having at least one child who is good and at least one child who is wise. The latter, on the other hand, represents the set of individuals with at least one child who is good and wise. Since for any two sets S_1 and S_2 the set S_1 is included in the set S_2 if and only if the intersection of S_1 and the complement of S_2 is an empty set, then we have to examine if the concept $(\exists \mathit{HasChild}.\mathit{Good}) \sqcap (\exists \mathit{HasChild}.\mathit{Wise}) \sqcap \neg\exists \mathit{HasChild}.(\mathit{Good} \sqcap \mathit{Wise})$ is unsatisfiable. After transforming it to NNF, the input concept in the tableau from Fig. 1 has the form $(\exists \mathit{HasChild}.\mathit{Good}) \sqcap (\exists \mathit{HasChild}.\mathit{Wise}) \sqcap \forall \mathit{HasChild}.(\neg \mathit{Good} \sqcup \neg \mathit{Wise})$. For the sake of brevity, the descriptions *HasChild*, *Good* and *Wise* are symbolized in the figure by the letters *H*, *G* and *W*, respectively.

Every node of the tableau is additionally labelled by a numerical identifier followed by a colon. Also, the tableau contains additional edge labels, each of them indicating the expansion rule applied to a parent node in order to obtain a child node. Moreover, for the sake of clarity, we assume that the multiple application of the same inference rule to subsequent nodes lying on the same path can be denoted by one edge with the label containing the number of applications of the given rule. For example, an edge with the label $\sqcap$-rule$\times 3$ can be replaced by the path representing a triple application of the $\sqcap$-rule to subsequent nodes. Furthermore, also for clarity reasons, we skip some of the already expanded assertions in a node label which do not participate in further inferences. It should be observed that the inclusion of concepts, considered in the example, does not hold since

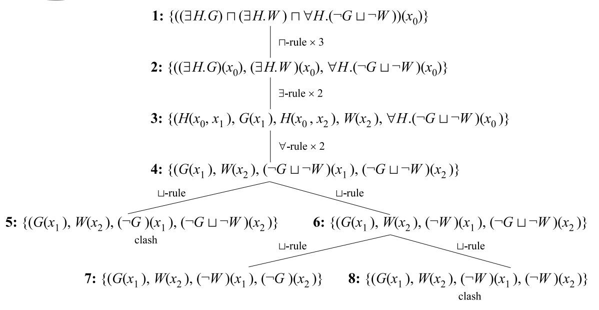

1: $\{((\exists H.G) \sqcap (\exists H.W) \sqcap \forall H.(\neg G \sqcup \neg W))(x_0)\}$
$\sqcap$-rule × 3
2: $\{((\exists H.G)(x_0), (\exists H.W)(x_0), \forall H.(\neg G \sqcup \neg W)(x_0)\}$
$\exists$-rule × 2
3: $\{(H(x_0, x_1), G(x_1), H(x_0, x_2), W(x_2), \forall H.(\neg G \sqcup \neg W)(x_0)\}$
$\forall$-rule × 2
4: $\{(G(x_1), W(x_2), (\neg G \sqcup \neg W)(x_1), (\neg G \sqcup \neg W)(x_2)\}$
$\sqcup$-rule
$\sqcup$-rule
5: $\{(G(x_1), W(x_2), (\neg G)(x_1), (\neg G \sqcup \neg W)(x_2)\}$
clash
6: $\{(G(x_1), W(x_2), (\neg W)(x_1), (\neg G \sqcup \neg W)(x_2)\}$
$\sqcup$-rule
$\sqcup$-rule
7: $\{(G(x_1), W(x_2), (\neg W)(x_1), (\neg G)(x_2)\}$
8: $\{(G(x_1), W(x_2), (\neg W)(x_1), (\neg W)(x_2)\}$
clash

Fig. 1. Open tableau for the example input concept.

the concept $(\exists \mathit{HasChild}.\mathit{Good}) \sqcap (\exists \mathit{HasChild}.\mathit{Wise})$ encompasses also individuals with no child who is good and wise. Instead, they have at least two children comprising one who is only good and one, who is only wise. In consequence, the input concept is satisfiable and the clash-free node 7 of the (open) tableau represents its model. One has to notice that all role assertions are omitted in the node for the sake of brevity, as has been said before. The model is the interpretation with $\Delta^{\mathcal{I}} = \{x_0, x_1, x_2\}$, $\mathit{HasChild}^{\mathcal{I}} = \{(x_0, x_1), (x_0, x_2)\}$, $\mathit{Good}^{\mathcal{I}} = \{x_1\}$ and $\mathit{Wise}^{\mathcal{I}} = \{x_2\}$.

It should be noted that the tableau calculus does not define any particular algorithm for the creation of the tableau. We discuss this issue in the next section.

3. Elements of the inference algorithm

When formulating an inference algorithm for the $\mathcal{ALC}$ tableau calculus, a procedure which builds a tableau requires considering two general issues.

1. The *computation rule*, determining the order in which assertions from the given node are chosen as premises for tableau rules.

2. The *search strategy*, i.e., the way the nodes of the tableau are selected for expansion.

A computation rule can also be regarded as a strategy for ordering the application of tableau rules. It can be proved (see, e.g. (Baader and Sattler, 2001)), that the soundness of the algorithm is independent of the computation rule. In other words, the reasoning procedure returns correct results for any rule that, however, may affect the reasoning complexity.

As for the search strategy, an important property of the $\mathcal{ALC}$ tableau calculus is that the whole knowledge necessary for node expansion or for clash detection is contained in a given node. Thus, there is no information exchange between nodes belonging to different branches. In consequence, branches of the tableau can be constructed independently of one another, particularly in parallel. Moreover, this is an example of the so-called *embarrassingly parallel* problem. It means that no particular effort is needed to segment the tableau and it can be done in many ways. The reasoning method described in this paper takes advantage of this property.

As has been stated before, we consider three simple computation rules. The first of them, called the *arbitrary* strategy (in short: A-strategy), assumes an arbitrary, "Prolog-like" order of premises. More precisely, a node label is represented as a list of assertions and the rule always selects the leftmost possible element. After the selected assertion is expanded by a relevant inference rule, the conclusions of the rule are monotonously added to the node label. As has been mentioned, the premises (i.e., expanded assertions) cannot be removed from the label, although the majority of them (excluding value restrictions) do not contribute any further conclusions. Nevertheless, they act as rule blockers, preventing (possibly infinite) reapplication of a rule to the same assertion.

This issue has been discussed by Herchenröder (2006), who pays attention to a particular computation rule in the satisfiability checking algorithm originally defined by Schmidt-Schauß and Smolka (1991). The method was also described by Baader *et al.* (2003), who called it the *trace* technique. Regarding this, we call the examined computation rule the *trace* strategy (in brief: T-strategy). The algorithm using the trace strategy is PSPACE-complete, unlike the procedure with the arbitrary rule, which has an exponential time and space complexity. The T-strategy imposes an ordering on the application of tableau rules. In particular, if more than one rule is relevant to the given node, then they are processed in the following order: (i) clash rules, (ii) $\sqcap$-rules and $\sqcup$-rules, (iii) the $\exists$-rule which has to be applied exhaustively to all relevant assertions in the node, (iv) the $\forall$-rule that should be handled like the $\exists$-rule. The trace strategy has a crucial feature—every concept assertion in the node can be expanded at most once. In other words, no inference rule can be applied to it again in any successor of the node.

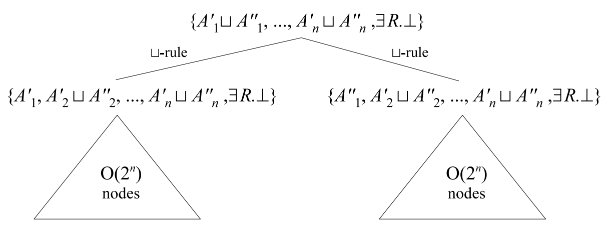
This approach yields two advantages except the reduction of space complexity (Baader *et al.*, 2003; Herchenröder, 2006). First, expansion rules can be simplified by skipping the conditions (b) in their definitions, since no blocking is necessary. This is particularly convenient with regard to the implementation of rules in the lean deduction style. Second, every assertion to which an inference rule has been applied can be removed from a node label as useless in the further reasoning process. This decreases the cardinality of a label and therefore it should reduce computational time. Moreover, a label can be additionally reduced by the following, less obvious optimisation. Let us assume a state when the $\exists$-rule is to be applied to the given node. This means that the node contains no clash and no other concept assertions than atomic ones or those with existential quantification or with the value restriction constructor. One should notice that the application of the $\exists$-rule always produces concept assertions with new individuals and the $\forall$-rule is possibly relevant only to these newly created assertions. Hence, new assertions cannot form any complementary pair with present atomic assertions, even after expansion. In consequence, all the latter assertions can be deleted from the node since they do not contribute in any way to further deduction. Also, the $\exists$-rule may be combined with the $\forall$-rule in the new $\exists\forall$-rule (Baader *et al.*, 2003) of the form given below:

$\exists\forall$-rule: if $S_{\exists R,x} \neq \emptyset$ then
$$\mathcal{A}' = \mathcal{A} \setminus S_{\exists\forall R,x} \cup \{C(y) | S \in S_{R,x} \wedge C \in S\},$$

where $S_{\exists R,x} = \{C | (\exists R.C)(x) \in \mathcal{A}\}$, $S_{\forall R,x} = \{C | (\forall R.C)(x) \in \mathcal{A}\}$, $S_{\exists\forall R,x} = S_{\exists R,x} \cup S_{\forall R,x}$ and $S_{R,x} = \{\{C\} \cup S_{\forall R,x} | C \in S_{\exists R,x}\}$. The symbol y denotes a new individual which does not occur in the node before and is unique in every element $C(y)$ of the defined set. The remaining expansion rules, in turn, are specified as follows:

$\sqcap$-rule: if $(C_1 \sqcap C_2)(x) \in \mathcal{A}$ then
$$\mathcal{A}' = \mathcal{A} \setminus (C_1 \sqcap C_2)(x) \cup \{C_1(x), C_2(x)\}$$
$\sqcup$-rule: if $(C_1 \sqcup C_2)(x) \in \mathcal{A}$ then
$$\mathcal{A}' = \mathcal{A} \setminus (C_1 \sqcup C_2)(x) \cup \{C_1(x)\},$$
$$\mathcal{A}'' = \mathcal{A} \setminus (C_1 \sqcup C_2)(x) \cup \{C_2(x)\}.$$

Unfortunately, the trace strategy has a drawback, which is manifested particularly in case of unsatisfiable concepts. This is a known effect (Baader *et al.*, 2003) that may occur during the expansion of nodes containing unions. It can be illustrated by the following example. Let us assume the node label $\{A'_1 \sqcup A''_1, \ldots, A'_n \sqcup A''_n, \exists R.\bot\}$. For the sake of clarity, we skipped individuals in assertions leaving just concept descriptions. We also assume that all concepts A'_i and A''_i are satisfiable for $i = 1, \ldots, n$. Nevertheless, the label contains a clash since the concept $\exists R.\bot$ is equivalent to $\bot$. This contradiction could be detected in two inference steps, namely, by subsequent application of the $\exists$-rule and the *clash*1-rule. However, the trace strategy processes existential quantifications only after all unions are expanded. This leads to unnecessary creation of 2^n clash nodes, as depicted in Fig. 2. Every clash node has a direct ancestor of the form



$\{A'_1 \sqcup A''_1, \ldots, A'_n \sqcup A''_n, \exists R.\bot\}$
$\sqcup$-rule
$\sqcup$-rule
$\{A'_1, A'_2 \sqcup A''_2, \ldots, A'_n \sqcup A''_n, \exists R.\bot\}$
$\{A''_1, A'_2 \sqcup A''_2, \ldots, A'_n \sqcup A''_n, \exists R.\bot\}$
O(2^n) nodes
O(2^n) nodes

Fig. 2. Tableau for the root $\{A'_1 \sqcup A''_1, \ldots, A'_n \sqcup A''_n, \exists R.\bot\}$ and the T-strategy.

$\{A_1, \ldots, A_n, \exists R.\bot\}$, where A_i equals either A'_i or A''_i for $i = 1, \ldots, n$. Hence, in the worst case, the reasoning algorithm with the trace strategy performs $O(2^n)$ times more inference steps then the method with arbitrary selection of premises if the latter starts from the assertion with the concept $\exists R.\bot$. This effect, in some cases, lengthens computational time even by a few orders of magnitude. Admittedly, it can be reduced by pruning the tableau using various optimisation techniques, for example, *backjumping* (Baader *et al.*, 2003). However, all these methods work on global data structures, which breaks the embarrassingly parallel nature of the reasoning process.

Regarding this, we define a simple computation rule called the *delayed branching* strategy (for brevity, DB-strategy). It states that the $\sqcup$-rule can be applied to the node only if no other rule is relevant to it. The ordering of the rest of tableau rules is the same as in the arbitrary approach. The DB-strategy does not support many convenient features of the T-strategy, namely, in general, it requires the blocking of inference rules and assertions cannot be removed from tableau nodes in the reasoning process (except some cases that are handled by optimisation presented in Section 5). However, this strategy reduces the disadvantageous effect described above and can be implemented with no particular effort in the parallel lean reasoner. The strategy behaves surprisingly well in comparison with the trace strategy (and also to the arbitrary strategy) in tests whose results are given in Section 6.

4. $\mathcal{ALC}$ DL reasoning in the relational model

In order to make the implementation of the tableau calculus independent of the search strategy, we express it in the relational model in the Oz language. In this approach, a program is a sequence of statements (in particular, procedure calls) which can either cease normally and produce results (namely, *answers*) or terminate by *failure* (i.e., with no answer). A program can also create a *choice point*, which causes the forking of computations into independent paths producing alternative answers.

A program that is to be run should be stored in a *computation space* (Van Roy and Haridi, 2004; Schulte, 2000). A space, among other properties, encapsulates computations so that they are separated form the exterior, particularly from processes performed in other spaces. The set of operations defined on spaces comprises the creation of a space for a new process as well as the merging, cloning and killing of spaces. These operations are performed by search engines. A space can communicate with an engine by appropriate statements. In particular, the statement `{Space.choose N}`, executed in the given space, tells the engine to create a choice point with `N` alternatives. The engine in the response clones the space `N` times and sends to each copy a numerical identifier ranging from 1 to `N`. The identifier becomes a value of the expression `{Space.choose N}`. The process of subsequent creation of spaces results in a *search tree*. Every leaf of the fully expanded tree is either a *solved* or a *failed* space. A solved space contains the result of normally terminated computations. It should be noted that computations executed inside spaces determine the shape and the content of the search tree. However, they do not settle the order in which spaces are created and processed—this depends upon the search engine only. In this way the declarative semantics of the program, corresponding to the structure of the tree, can be separated from the operational semantics represented by the search strategy.

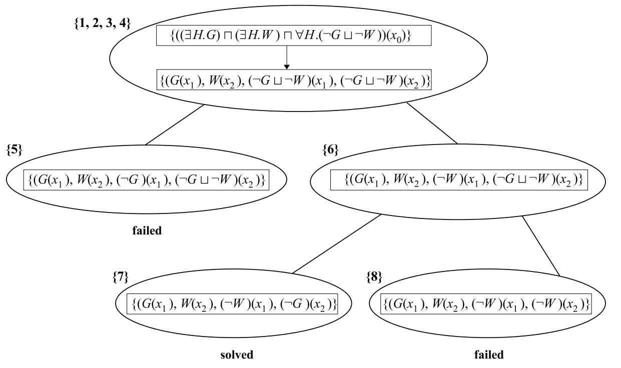

{1, 2, 3, 4}
{((∃H.G) ⊓ (∃H.W) ⊓ ∀H.(¬G ⊔ ¬W))(x0)}
{(G(x1), W(x2), (¬G ⊔ ¬W)(x1), (¬G ⊔ ¬W)(x2)}
{5}
{(G(x1), W(x2), (¬G)(x1), (¬G ⊔ ¬W)(x2)}
failed
{6}
{(G(x1), W(x2), (¬W)(x1), (¬G ⊔ ¬W)(x2)}
{7}
{(G(x1), W(x2), (¬W)(x1), (¬G)(x2)}
solved
{8}
{(G(x1), W(x2), (¬W)(x1), (¬W)(x2)}
failed

Fig. 3. Search tree for the tableau from Fig. 1.

We use a search tree as a representation of a tableau. Regarding this, we consider the assumptions given below. The symbol $\mathcal{A}$ denotes any internal tableau node while $S_{\mathcal{A}}$ stands for the space in which the node $\mathcal{A}$ is computed.

1. The root space corresponds to the root of a tableau.

2. If $\mathcal{A}'$ is the only one direct successor of the node $\mathcal{A}$, created by any expansion rule different from the $\sqcup$-rule, then $\mathcal{A}'$ replaces $\mathcal{A}$ in the space $S_{\mathcal{A}}$.

3. If nodes $\mathcal{A}'_1, \ldots, \mathcal{A}'_n$ are direct successors of the node $\mathcal{A}$ obtained by an application of the $\sqcup$-rule, then for every node $\mathcal{A}'_i$ a new space $S_{\mathcal{A}'_i}$ is created, which is a direct successor of the space $S_{\mathcal{A}}$ in the search tree, for $i = 1, \ldots, n$.

4. A clash-free node is mapped to a solved space. The result of computations can possibly represent a model of the input concept.

5. A clash node is represented by a failed space.

The correctness of the second assumption follows from the fact that every node $\mathcal{A}$ of the tableau having the only one successor $\mathcal{A}'$ does not participate in any further inferences after the creation of the node $\mathcal{A}'$, and thus it can be replaced by $\mathcal{A}'$ in the tableau construction process. In Fig. 3 we show a search tree for the tableau from Fig. 1. The upper and the lower rectangle in a space encloses a label of the starting and, respectively, the resulting node created in this space. If both the nodes are identical , then they are depicted by one rectangle. Each space in the tree is additionally labelled by a set of numbers which identify tableau nodes corresponding to the space.

The Mozart programming system provides various library classes of engines implementing different search strategies. In particular, instances of the class `Search.parallel` are parallel search engines designed to work on distributed machines. The engine can be regarded as a team of concurrent autonomous agents comprising a *manager* and a group of *workers*. The manager controls the computations by finding a work for idle workers and collecting the results, whereas the workers construct fragments of the search tree. Members of the team communicate by exchanging messages. A detailed description of this architecture, including the communication protocol, is to be found in the work of Schulte (2000). Below, we give an example of the statement creating a new parallel engine.

```
Eng = {New Search.parallel
        init(w1:2\#ssh w2:3\#ssh)}.
```

It consists of the manager (initiated locally) and five workers, started on remote computers via secure shell (`ssh`) commands—two on the machine `w1` and three on the other machine `w2`. One can tell the engine to execute the given procedure by the following statement:

$$\{\text{Eng } \mathit{ProcedureCall}\}.$$

The result of computations becomes a value of the expression written above. It should be remarked that the subexpression *ProcedureCall* is given in a pseudocode in order to skip some technical details.

5. Implementation of the reasoning system

A key part of the reasoning system is the procedure `Prove`. In this section we describe three variants of this procedure, each of them implementing a different computation rule. The system processes $\mathcal{ALC}$ expressions, which are represented by Oz data structures. Furthermore, we use subsequent numbers starting from 0 to encode individuals. Atomic descriptions (of concepts and roles) are expressed as *atoms* and complex concept descriptions as well as concept and role assertions are denoted by *tuples* (Van Roy and Haridi, 2004). The correspondence between the notation considered and the standard $\mathcal{ALC}$ syntax is given in Table 1. Primed symbols are Oz representations of their unprimed $\mathcal{ALC}$ counterparts—we use this convention also in the sequel. Expressions of the form $C_1 \sqcap \ldots \sqcap C_n$ and $C_1 \sqcup \ldots \sqcup C_n$ stand for nested intersections and unions, respectively. It should be noticed that the use of the negation constructor is restricted to atomic concepts, since all concept descriptions are assumed to be in NNF.

Table 1. Oz representation of $\mathcal{ALC}$ expressions.

$\mathcal{ALC}$ syntax	Oz notation
$\top$	`top`
$\bot$	`bot`
$\neg A$	`neg(`A'`)`
$C_1 \sqcap \ldots \sqcap C_n$	`and(`C'_1 `...` C'_n`)`
$C_1 \sqcup \ldots \sqcup C_n$	`alt(`C'_1 `...` C'_n`)`
$\exists R.C$	`ex(`R' C'`)`
$\forall R.C$	`all(`R' C'`)`
$C(x)$	C'`#`x'

The first variant of the procedure `Prove`, given in Fig. 4, realises the A-strategy. It is a straightforward implementation of the computation rule described in Section 3. The procedure constructs a tableau for a given concept description and checks whether the description is satisfiable, that is to say, if the tableau contains any open branch. If so, the argument `Model` is bound to the list representing a model of the input concept. The list encloses role assertions as well as assertions with literals included in a given clash-free node. Otherwise, when the tableau is closed, the execution of the procedure results in a failure. In every subsequent call, the procedure handles one node of the tableau and processes it in the *current space*. For the sake of efficiency, a node label is split into two disjoint subsets containing concept assertions and, role assertions. The first one is represented by the list `Concs`, while the latter by the list `Roles`. Every element of the list `Roles` has the form R'`#`x'`#[`y'_1 `...` y'_n`]`, which corresponds to the following set of role assertions: $\{R(x, y_1), \ldots, R(x, y_n)\}$. This representation is motivated by practical reasons, namely, by the way the $\forall$-rule is applied and implemented.

In order to speed up the detection of clashes, assertions with positive and negative literals from the list `Concs` are additionally stored in two lists, namely, `PLits` and `NList`, respectively. The subsequent argument of the procedure `Prove`, i.e., `I`, is a number standing for the individual which has been added to the current branch as the last one. This number, increased by one, acts as a unique name for a new concept or role instance, when it is introduced to the tableau. The argument `N` is used for the detection of a clash-free node. It indicates the current number of assertions from the list `Concs` to which no expansion rule can be applied.

The reasoning process starts from the execution of the procedure `Prove` with the following actual arguments:

{`Prove` [C'#0] `nil nil nil 0 0 Model`},

where the symbol C' stands for the input concept and the symbol `nil` denotes an empty list. In every subsequent call, the procedure takes the first element `Conc` of the list `Concs` and tries to apply a relevant tableau rule to it. If this is not possible (because the rule is blocked), the element is moved to the end of the list and the procedure is recursively called with the argument `N` incremented by 1 (line 37). If the value of this argument is equal to the length of the list `Concs`, then the current node is regarded as clash free since no rule can be applied to it. In such a case, the argument `Model` is bound to the representation of the model of the input concept (line 4).

Otherwise, namely, when `Conc` is a concept assertion to which a tableau rule can possibly be applied, the procedure `Prove` first checks whether it can be the $clash1-$rule (line 7) or the $clash2-$rule (lines 8–15). The latter is implemented in two variants for assertions with positive and negative literals, respectively. The execution of the clash rules results in failure in the current space, which is caused by the statement `fail`. In the other case, namely, when there is no complementary counterpart for the assertion `Conc`, it is moved to the end of the list `Concs` in the next call of the procedure `Prove` (lines 10 and 14).

Lines 16–20 implement the $\sqcap$-rule. If the assertion `Conc` is an intersection, then it is decomposed into elements and those which are not members of the list `Concs` (lines 18–19) are added to this list (line 20). If there are no such elements, then the rule is blocked. The implementation of the $\sqcup$-rule is given in lines 21–25. It breaks the assertion `Conc` (containing a union) into components which are collected on the list `L` (lines 22–23). If none of the components is a member of the list `Concs`, then for each of them a clone of the current space is created by the statement {`Space.choose` {`Length L`}} (line 25). In every clone, this expression evaluates to a distinct number, which is used in turn by the function `Nth` to select the respective element from the list `L`. This component is added to the list `Concs` in the subsequent call of the procedure `Prove`.

Lines 26–31 correspond to the $\exists$-rule, relevant to an assertion of the form $(\exists R.C)(x)$. The execution of this rule generally consists in inserting appropriate new elements to lists `Concs` and `Roles` in the next call of the procedure `Prove` (line 31). However, it should be noticed that no new elements are added to the list `Concs` if the concept C equals $\top$. This follows from the fact that assertions with the most general concept are unessential for the reasoning process and therefore can be ignored.

Finally, in lines 32–35, the $\forall$-rule is implemented. The execution of this rule (line 35) takes place if the list `Concs` contains an assertion of the form $(\forall R.C)(x)$ and the list `Roles` includes the assertion `C.1#X#Inds`, where `C.1` and `X` stand for R and x, respectively, and `Inds` is a nonempty list of fillers of the role R for x (lines 32–33). Furthermore, the rule is not executed if C is the most general concept (i.e., $\top$), for the same reason as in

```
proc {Prove Concs Roles PLits NLits I N Model}                                    % 1
   if N == {Length Concs}                                                         % 2
   then                                                                           % 3
      {Flatten [PLits NLits Roles] Model}                                         % 4
   else                                                                           % 5
      Conc = Concs.1 C = Conc.1 X = Conc.2 Op = {Label C} L Inds Roles1 in        % 6
      if C == bot orelse C == neg(top) then fail                                  % 7
      elseif {IsAtom C} andthen {Not {Member Conc PLits}} then                    % 8
         if {Member Conc NLits} then fail else                                    % 9
            {Prove {Append Concs.2 [Concs.1]} Roles Conc|PLits NLits I 0 Model}   % 10
         end                                                                      % 11
      elseif Op == neg andthen {Not {Member C.1#X NLits}} then                    % 12
         if {Member C.1#X PLits} then fail else                                   % 13
            {Prove {Append Concs.2 [Concs.1]} Roles PLits (C.1#X)|NLits I 0 Model}      % 14
         end                                                                      % 15
      elseif Op == and andthen                                                    % 16
         (L={Filter {Map {Record.toList C} fun {$ E} E#X end}                     % 17
                    fun {$ E} {Not {Member E Concs.2}} end}) \= nil               % 18
      then                                                                        % 19
         {Prove {Append L Concs} Roles PLits NLits I 0 Model}                     % 20
      elseif Op == alt andthen                                                    % 21
            {Not {Some (L={Map {Record.toList C} fun {$ E} E#X end})              % 22
                        fun {$ E} {Member E Concs.2} end}}                        % 23
      then                                                                        % 24
         {Prove {Nth L {Space.choose {Length L}}}|Concs Roles PLits NLits I 0 Model}    % 25
      elseif Op == ex andthen                                                     % 26
            ({Not {SelRol Roles C.1#X#Inds Roles1}} orelse (C.2 \= top andthen    % 27
             {Not {Some Concs fun {$ E} {And (E.1 == C.2) {Member E.2 Inds}} end}}))    % 28
      then                                                                        % 29
         if C.2 == top then L = Concs else L = C.2#(I+1)|Concs end                % 30
         {Prove L C.1#X#(I+1|Inds)|Roles1 PLits NLits I+1 0 Model}                % 31
      elseif Op == all andthen C.2 \= top andthen {SelRol Roles C.1#X#Inds _} andthen   % 32
            (L={Filter Inds fun {$ E} {Not {Member C.2#E Concs.2}} end}) \= nil   % 33
      then                                                                        % 34
         {Prove {Append {Map L fun {$ E} C.2#E end} Concs} Roles PLits NLits I 0 Model} % 35
      else                                                                        % 36
         {Prove {Append Concs.2 [Concs.1]} Roles PLits NLits I N+1 Model}         % 37
      end                                                                         % 38
   end                                                                            % 39
end                                                                               % 40
```

Fig. 4. Definition of the procedure `Prove` with the A-strategy.

the case of the $\exists$-rule.

The second variant of the procedure `Prove`, which implements the T-strategy as the computation rule, is presented in Fig. 5.

The procedure does not use the parameter `N` (present in the previous version) since no inference rule can be applied to any concept assertion more than once. Also, unlike the previous version, the procedure does not construct a model of the input concept. Instead, it assigns the value `sat` to the argument `Result` if the concept is satisfiable. The parameter `Roles` is a list of tuples of the form R'`#`x'`#[`C'_1 `...` C'_m`]#[`D'_1 `...` D'_n`]`, which corresponds to the set of assertions $(\exists R.C_1)(x), \dots, (\exists R.C_m)(x)$ and $(\forall R.D_1)(x), \dots, (\forall R.D_n)(x)$. This representation follows from the way the $\exists\forall$-rule is implemented.

The execution of the procedure consists of two general steps, which reflects the ordering imposed by the T-strategy. At first, assertions from the list `Concs` are processed (lines 1–23). In particular, clashes, intersections and unions are treated similarly as in the procedure from Fig. 4. However, unlike in that case, premises are deleted from the list `Concs` after each reasoning step.

Assertions of the form $(\exists R.C)(x)$ and $(\forall R.C)(x)$ are not handled by inference rules at this stage, but they are moved to the list `Roles`. This is done by two external procedures, namely, `ToRoleEx` (line 20) and `ToRoleAll` (line 22), which transform the assertions considered to the representation used on the target list. In the next step, to wit, when the list `Concs` is empty, the $\exists\forall$-rule is applied to every element of the list `Roles` (lines 25–28). This results in the construction of a new content of the list `Concs`, which is processed in the subsequent call of the procedure `Prove` (lines 27–28). The reasoning process finishes when either the current node contains a clash (lines 4, 6 and 9) or the lists `Concs` and `Roles` are both empty. In the latter case, the value `sat`, bound to the argument `Result`, is returned (line 29).

Finally, we describe the third variant of the procedure `Prove` given in Fig. 6, which implements the DB-strategy. It handles all concept assertions except unions in a similar way as the procedure from Fig. 4. Reasoning about unions, on the other hand, is suspended until expressions of all other types are processed. More precisely, assertions with unions are initially moved to the list `Alts` (lines 15–16), which is an additional argument of the procedure `Prove`. Elements of this list are expanded as the last ones (lines 24–26). Also, we somehow optimised the realisation of the $\exists$-rule and the $\forall$-rule. For this purpose, role assertions being elements of the list `Roles` are represented as tuples of the form R'`#`x'`#[`y'_1 `...` y'_m`]#[`C'_1 `...` C'_n`]`. The elements $y_1, \dots, y_m$ are fillers of the role R for x coming from the processing of concept assertions $(\exists R.C)(x)$ belonging to the list `Concs`. The concepts $C_1, \dots, C_n$, in turn, enter the list as results of the expansion of value restrictions $(\forall R.C)(x)$. Furthermore, when an expression $(\exists R.C)(x)$ is expanded, the new filler y is added

```
proc {Prove Concs Roles PLits NLits I Result}                                         % 1
   if Concs \= nil then                                                               % 2
      Conc = Concs.1 C = Conc.1 X = Conc.2 Lab = {Label C} L in                      % 3
      if C == bot orelse C == neg(top) then fail                                      % 4
      elseif {IsAtom C} andthen C \= top andthen {Not {Member Conc PLits}} then       % 5
       if {Member Conc NLits} then fail                                               % 6
       else {Prove Concs.2 Roles Conc|PLits NLits I Result} end                      % 7
      elseif Lab == neg andthen {Not {Member C.1#X NLits}} then                       % 8
       if {Member C.1#X PLits} then fail                                              % 9
       else {Prove Concs.2 Roles PLits (C.1#X)|NLits I Result} end                   % 10
      elseif Lab == and andthen                                                       % 11
         (L={Filter {Map {Record.toList C} fun {$ E} E#X end}                         % 12
                    fun {$ E} {Not {Member E Concs.2}} end}) \= nil then              % 13
         {Prove {Append L Concs.2} Roles PLits NLits I Result}                        % 14
      elseif Lab == alt andthen                                                       % 15
            {Not {Some (L={Map {Record.toList C} fun {$ E} E#X end})                  % 16
                        fun {$ E} {Member E Concs.2} end}} then                       % 17
         {Prove {Nth L {Space.choose {Length L}}}|Concs.2 Roles PLits NLits I Result}  % 18
      elseif Lab == ex then                                                           % 19
         {Prove Concs.2 {ToRoleEx Roles C.1 X C.2} PLits NLits I Result}              % 20
      elseif Lab == all then                                                          % 21
       {Prove Concs.2 {ToRoleAll Roles C.1 X C.2} PLits NLits I Result}               % 22
      else {Prove Concs.2 Roles PLits NLits I Result} end                             % 23
   elseif Roles \= nil then                                                           % 24
      NCs={FoldR Roles fun {$ _#_#Es#As T} {Append {Map Es fun {$ E} E|As end} T} end nil} % 25
   in                                                                                 % 26
      {Prove {Flatten {List.mapInd NCs fun {$ J Cs} {Map Cs fun {$ C} C#I+J end} end}}  % 27
             nil nil nil I+{Length NCs} Result}                                       % 28
   else Result = sat end                                                              % 29
end                                                                                   % 30
```

Fig. 5. Definition of the procedure `Prove` with the T-strategy as the computation rule.

```
proc {Prove Concs Alts Roles PosLits NegLits I Res}                                   % 1
   if Concs \= nil then                                                               % 2
      Conc = Concs.1 C = Conc.1 X = Conc.2 Lab = {Label C} L in                      % 3
      if C == bot orelse C == neg(top) then fail                                      % 4
      elseif {IsAtom C} andthen C \= top andthen {Not {Member Conc PosLits}} then     % 5
       if {Member Conc NegLits} then fail                                             % 6
       else {Prove Concs.2 Alts Roles Conc|PosLits NegLits I Res} end                % 7
      elseif Lab == neg andthen {Not {Member C.1#X NegLits}} then                     % 8
       if {Member C.1#X PosLits} then fail                                            % 9
       else {Prove Concs.2 Alts Roles PosLits (C.1#X)|NegLits I Res} end             % 10
      elseif Lab == and andthen                                                       % 11
         (L={Filter {Map {Record.toList C} fun {$ E} E#X end}                         % 12
                    fun {$ E} {Not {Member E Concs.2}} end}) \= nil then              % 13
         {Prove {Append L Concs.2} Alts Roles PosLits NegLits I Res}                  % 14
      elseif Lab == alt andthen {Not {Member Conc Alts}} then                         % 15
         {Prove Concs.2 Conc|Alts Roles PosLits NegLits I Res}                        % 16
      elseif Lab == ex then CAlls RolesN in                                           % 17
         {ToRoleEx Roles C.1 X I CAlls RolesN}                                        % 18
         {Prove C.2#I|{Append CAlls Concs.2} Alts RolesN PosLits NegLits I+1 Res}     % 19
      elseif Lab == all then Exs RolesN in                                            % 20
         {ToRoleAll Roles C.1 X C.2 Exs RolesN}                                       % 21
       {Prove {Append Exs Concs.2} Alts RolesN PosLits NegLits I Res}                 % 22
      else {Prove Concs.2 Alts Roles PosLits NegLits I Res} end                       % 23
   elseif Alts \= nil then                                                            % 24
      L={Map {Record.toList Alts.1.1} fun {$ E} E#Alts.1.2 end} in                    % 25
      {Prove [{Nth L {Space.choose {Length L}}}] Alts.2 Roles PosLits NegLits I Res}  % 26
   else Res=Concs end                                                                 % 27
end                                                                                   % 28
```

Fig. 6. Definition of the procedure `Prove` with the DB-strategy as the computation rule.

to the list `[`y'_1 `...` y'_m`]` and the sequence of assertions $C_1(y), \ldots, C_n(y)$ is appended to the list `Concs` (lines 17–19). The processing of a value restriction $(\forall R.C)(x)$ (lines 20–22), on the other hand, results in adding the new concept C to the list `[`C'_1 `...` C'_n`]`. Also, the list of assertions `[`$C(y_1)'$ `...` $C(y_m)'$`]`, called `Exs`, is constructed by the external procedure `ToRoleAll` (line 21). Elements of this list extend the list `Concs` in the next call of the procedure `Prove` (line 22). It should be observed that the approach considered offers an advantage in that none of the elements of the list `Concs` has to be reused after the expansion and therefore can be removed from the list. Hence, the condition of the termination of the reasoning process can be defined as emptiness of both the lists `Concs` and `Alts`. In other words, no additional argument `N` is necessary as in the case of the procedure from Fig. 4.

6. Evaluation

In this section we present and analyse results of experiments intended for the comparison of the computation rules described in the previous sections. The computational environment consisted of a machine which processed the manager (Pentium-M 760, 2.0 GHz, 1 GB RAM, 1GBit Ethernet, Windows XP Home 2002) and up to five identical machines (Pentium P4D, 3.4 GHz, 1 GB RAM, 1 GBit Ethernet, Windows 2000 Professional 5.00) processing one worker each. If it does not lead to a confusion, in the sequel we often identify a machine processing a worker with this worker. All the computers were powered by the 1.3.2. Mozart system.

The testing data come from the benchmark set T98-sat (Horrocks and Patel-Schneider, 1998) encompassing nine types of concepts. Each of them is given

Table 2. T98-sat test results for the PS and DB strategies.

Problem	A	T	DB	Problem	A	T	DB
k_branch_n	1	1	1	k_branch_p	1	1	1
k_d4_n	1	2	3	k_d4_p	2	2	3
k_dum_n	8	21	21	k_dum_p	5	3	6
k_grz_n	21	21	21	k_grz_p	1	0	3
k_lin_n	5	7	21	k_lin_p	17	5	21
k_path_n	5	6	6	k_path_p	1	1	2
k_ph_n	5	4	3	k_ph_p	3	3	3
k_poly_n	5	21	21	k_poly_p	13	8	21
k_t4p_n	0	1	1	k_t4p_p	0	0	2

in both a satisfiable and an unsatisfiable variant, which results in 18 files. The letter ending a file name indicates the variant of the file contents; moreover, n stands for satisfiable concepts while p denotes unsatisfiable ones. Every file contains 21 numbered concept examples of increasing complexity. Furthermore, the computational time is expected to grow exponentially with a subsequent concept example. The testing method consists in finding the number of the most complex example which can be evaluated in no more than 100 seconds. We use this method, running the reasoning system on one machine, in order to compare the A, T and DB strategies in terms of absolute values. All problems were initially transformed to NNF. The results of tests are collected in Table 2; the number 0 means that no example of the concept can be evaluated in the given time limit.

When compared with the arbitrary strategy, the trace strategy provides better results for six types of concepts and worse outcomes in five cases. For seven concept types the results are the same. Also, it should be noticed that four out of five cases when the T-strategy performs worse concern unsatisfiable concepts. Furthermore, the strategy does not work better than the arbitrary one for any unsatisfiable concept. This is caused by the effect described in Section 3. Summing up, the "pure", unoptimised, T-strategy is only slightly more efficient than the A one. On the other hand, the DB-strategy appears considerably faster in the respected test than both remaining computation rules. It gives better results for eight concept types and at least as good outcomes as the other strategies for nine types of concepts. The DB-strategy performs somewhat worse only in one case.

In the next testing step, we selected some particular T98-sat concept examples to estimate the speedup obtained by executing the reasoning system in parallel with the increasing number of workers in the computational environment. The speedup is a quotient of problem solving time by one worker and by the given number of workers, respectively. Every concept example is denoted by its number in the file it belongs to, preceded by the file name. For example, the identifier k_path_p_6 represents the example number 6 in the file k_path_p. The examples were chosen under the general criterion that the time of computations performed by one worker has to range from 3 to 300 seconds. The relatively low left endpoint of this interval follows from difficulties in finding suitable testing data. Due to the exponential growth of computational time, only a few types of concepts have examples satisfying the given criterion. For every test, computational time is a system clock time taken as an arithmetic mean of five runs. Results obtained for satisfiable and unsatisfiable concepts differ significantly one from another and therefore are considered separately.

Table 3 contains computational time for satisfiable concepts processed by three respective strategies in the environment consisting of 1–5 workers. It should be reminded that the search engine stops after finding the first open branch. Furthermore, every worker constructs its part of the tableau in a depth-first manner from left to right. Also, at any branching point it can convey a part of work to any other worker, which is currently idle. In consequence, computational time strongly depends on the way the tableau is partitioned into subtrees. Moreover, if the first open branch is located close to the left-hand side of the tableau and all open branches are of similar length, then adding new machines cannot significantly speed up the reasoning process. This may explain why the computational time is nearly constant for problems processed by the T-strategy and for particular problems handled by the A-strategy (k_poly_n_5) and the DB-strategy (k_d4_n_3). Otherwise, namely, when the first open branch is located further from the left-hand side of the tableau, the computational time can shrink if this path occurs as the first one in any of the subtrees. The computational time may also increase if the tableau is partitioned in a less convenient way. This can be a source of such anomalies as a super-linear speedup or slowdown, which are observed in the case of the problems k_path_n_5 and k_t4p_n_1 for the arbitrary strategy and for the problem k_t4p_n_2 processed by the DB-strategy. Furthermore, it should be remarked that introducing new workers to the

Table 3. Computational time [s] for selected satisfiable T98-sat concepts.

Strategy	Problem	1 worker	2 workers	3 workers	4 workers	5 workers
A	k_lin_n_6	142.56	142.7	145.62	147.06	150.29
	k_path_n_5	97.2	67.72	81.37	70.31	72.45
	k_poly_n_5	83.83	82.74	82.83	82.88	82.31
	k_poly_n_6	178.17	178.9	182.38	183.45	183.31
	k_t4p_n_1	142.75	25.1	25.77	25.29	26.98
T	k_lin_n_7	21.53	19.88	19.89	18.73	19.01
	k_poly_n_21	14.28	14.25	14.35	14.29	14.26
	k_ph_n_4	10.18	10.08	10.05	10.02	9.98
	k_t4p_n_1	7.84	7.89	7.89	7.89	7.88
DB	k_d4_n_3	3.88	3.89	3.89	3.93	3.89
	k_path_n_6	57.91	60.25	58.34	47.05	43.28
	k_poly_n_17	32.75	30.98	30.82	30.99	30.57
	k_t4p_n_2	273.84	274.72	244.17	244.79	246.92

Table 4. Speedup for selected unsatisfiable T98-sat concepts.

Strategy	Problem	2 workers	3 workers	4 workers	5 workers
A	k_dum_p_5	1.98	2.92	3.87	4.73
	k_grz_p_1	1.97	2.92	3.80	4.68
	k_poly_p_12	1.91	2.56	2.81	3.39
	k_poly_p_13	1.83	1.97	2.63	3.88
	k_poly_p_14	1.82	2.75	2.91	4.34
T	k_dum_p_3	2.00	2.94	3.87	4.79
	k_lin_p_5	1.97	2.85	3.80	4.57
	k_poly_p_8	1.96	2.86	3.79	4.66
	k_poly_p_9	1.97	2.94	3.88	4.76
DB	k_dum_p_3	1.98	2.91	3.83	4.72
	k_grz_p_4	2.00	2.98	3.92	4,87
	k_ph_p_3	1.96	2.89	3.83	4.68
	k_poly_p_21	1.04	1.04	1.04	1.03

environment increases timing costs of network communication among parts of the search engine. In certain cases costs can exceed benefits coming from the partitioning of the search tree. This may be a reason why computational time systematically grows after connecting new workers for the problems `k_lin_n_6` and `k_poly_n_6` in the case of the A-strategy.

The other group of testing problems consists of unsatisfiable concepts only, for which the whole tableau has to be created since it does not contain any open branch. Hence, adding new machines to the environment nearly always yields a speedup, as confirmed by the results presented in Table 4.

As can be noticed, in the majority of cases the speedup is nearly linear. For example, connecting the subsequent worker (i.e., the second, the third, the fourth and the fifth one) to the computational environment, which processes the problem `k_dum_p_5` by the A-strategy, results in the shortening of the computational time by 1.98, 2.92, 3.87 and 4.73, respectively. However, for concepts of the type `k_poly_p`, processed by the same strategy, the speedup apparently fluctuates. This may follow from the fact that the distribution of the reasoning process depends upon the structure of the search tree. In particular, trees for the examined class of concepts are highly unbalanced and therefore workers can be loaded unevenly. This issue requires more thorough observations.

Also, a concept of the same type, namely, `k_poly_p_21`, produces exceptional results for the strategy DB, i.e., it practically yields no speedup. The explanation of this effect lies in the behaviour of the DB-strategy, which strongly restricts the branching of the tableau for the `k_poly_p` concepts compared with the T-strategy. Moreover, the tableau constructed by the strategy T for the problem `k_poly_p_9`, which is a simpler version of the problem `k_poly_p_21`, contains nearly 1.7 million of branches, while the number of branches for the latter concept example, processed by the strategy DB, equals only 313. This number is too small to cause any observable speedup. It should be remarked that an “eager” branching, performed by the T-strategy, on the one hand enhances the possibility of the creation of the tableau in parallel, although on the other hand it often augments the negative effect concerning unnecessary copying of clash nodes, described in Section 3. Summing up, the strategy DB behaves generally better than the remaining strategies in experiments discussed in this section. All the strategies produce a similar speedup, but the DB approach frequently results in a notably shorter computational time.

7. Final remarks

In this paper, we experimentally analysed three simple computation rules, namely the A-strategy, the T-strategy and the DB-strategy, in a tableau-based, parallel lean reasoning system for the $\mathcal{ALC}$ description logic. The system is implemented in the relational programming model in the Oz language and executed by the parallel search engine on distributed machines. Empirical evaluation of the rules, performed on the testing data coming from the benchmark set T98-sat, gives similar outcomes for the speedup. However, the DB-strategy is generally more efficient than the remaining computation rules, since for the majority of tests it results in significantly shorter computational times. Nevertheless, a comprehensive analysis of the issue considered requires more tests, particularly on realistic data. These tests are intended for the future.

Acknowledgment

The work was supported in part by the Poznań University of Technology under Grant No. DS 45-083/10 and in part by the Polish Ministry of Science and Higher Education under Grant No. N 516 369536. The author wishes to thank the anonymous reviewers for their constructive criticism including valuable remarks that helped improve the presentation of this paper.

References

Amir, E. and Maynard-Zhang, P. (2004). Logic-based subsumption architecture, *Artificial Intelligence* **153**(1–2): 167-237.

Aslani, M.and Haarslev, V. (2008). Towards parallel classifcation of TBoxes, *in* F. Baader, C. Lutz, and B. Motik (Eds.), *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, CEUR Workshop Proceedings, Vol. 353.

Baader, F., McGuinness, D., Nardi, D. and Patel-Schneider, P. (Eds.) (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press, Cambridge.

Baader, F. and Sattler, U. (2001). An overview of tableau algorithms for description logics, *Studia Logica* **69**(1): 5–40.

Beckert, B. and Possega, J. (1995). lean$T^{A}P$: Lean, tableau-based deduction, *Journal of Automated Reasoning* **15**(3): 339–358.

Calvanese, D., Lenzerini, M. and Nardi, D. (1999). Unifying class-based representation formalisms, *Journal of Artificial Intelligence Research* **11**: 199–240.

De Giacomo, G., Iocchi, L., Nardi, D. and Rosati R. (1996). Moving a robot: The KR&R approach at work, *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning, Cambridge, MA, USA*, pp. 198–209.

Devanbu, P. and Jones, M. (1997). The use of description logics in KBSE systems, *ACM Transactions on Software Engineering and Methodology* **6**(2): 141–172.

Herchenröder, T. (2006). *Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics*, M.Sc. thesis, University of Edinburgh, Edinburgh.

Horrocks, I. and Patel-Schneider, P.F. (1998). DL systems comparison (summary relation), *Proceedings of the 1998 International Workshop on Description Logics (DL'98)*, CEUR Workshop Proceedings, Vol. 11, pp. 55–57.

Hustadt, U., Motik, B. and Sattler, U. (2004). Reducing SHIQ-description logic to disjunctive datalog programs, *in* D. Dubois, C.A. Welty and M.-A. Williams (Eds.), *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, AAAI Press, Menlo Park, CA, pp. 152–162.

Liebig, T. and Müller, F. (2007). Parallelizing tableaux-based description logic reasoning, *in* R. Meersman, Z. Tari, and P. Herrero (Eds.), *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops,* Lecture Notes in Computer Science, Vol. 4806, Springer-Verlag, Berlin/Heidelberg, pp. 1135–1144.

Meissner, A. (2009a). Introducing parsimonious rules to a parallel reasoning system for the $\mathcal{ALC}$ description logic, *Proceedings of the 7th Conference on Computer Methods and Systems, CMS'09, Cracow, Poland,* pp. 75–80.

Meissner, A. (2009b). A simple parallel reasoning system for the $\mathcal{ALC}$ description logic, *in* N.T. Nguyen, R. Kowalczyk, and S.-M. Chen (Eds.), *Computational Collective Intelligence. Semantic Web, Social Networks and Multiagent Systems: Proceedings of the First International Conference, ICCCI 2009*, Lecture Notes in Artificial Intelligence, Vol. 5796, Springer-Verlag, Berlin/Heidelberg, pp. 413–424.

OWL Web Ontology Language Overview (2004), `http://www.w3.org/TR/owl-features/`.

Rector A.L., Zanstra, P., Solomon, W., Rogers, J., Baud, R., Ceusters, W., Claassen, A., Kirby, J., Rodrigues, J., Mori, A., van der Haring, E. and Wagner, J. (1998). Reconciling users' needs and formal requirements: Issues in developing a reusable ontology for medicine, *IEEE Transactions on Information Technology in Biomedicine* **2**(4): 229–242.

Rychtyckyj, N. (1996). DLMS: An evaluation of KL-ONE in the automobile industry, *in* L.C. Aiello, J. Doyle, and S.C.

Shapiro (Eds.), *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, Morgan Kaufmann, San Francisco, CA, pp. 588–596.

Schmidt-Schauß, M. and Smolka, G. (1991). Attributive concept descriptions with complements, *Artificial Intelligence* **48**(1): 1–26.

Schulte, C. (2000). *Programming Constraint Services*, Ph.D. thesis, Saarland University, Saarbrücken.

Semantic Web (2001). `http://www.w3.org/2001/sw/`.

The Mozart Programming System (2008). `http://www.mozart-oz.org`.

Tsarkov, D. and Horrocks, I. (2006). FaCT++ description logic reasoner: System description, *in* U. Furbach and N. Shankar (Eds.), *Automated Reasoning: Third International Joint Conference, IJCAR 2006*, Lecture Notes in Computer Science, Vol. 4130, Springer-Verlag, Berlin/Heidelberg, pp. 292–297.

Van Roy, P. and Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*, MIT Press, Cambridge, MA.

Wessel, M. and Möller, R. (2005). A high performance Semantic Web query answering engine, *in* I. Horrocks, U. Sattler and F. Wolter (Eds.), *Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, CEUR Workshop Proceedings, Vol. 147.

Adam Meissner received the Ph.D. degree from the Poznań University of Technology in 1999. Since 2000 he has been an assistant professor at the Institute of Control and Information Engineering of the same university. Dr. Meissner is an author or a co-author of over 30 papers in the field of artificial intelligence. His current research interests include distributed reasoning systems and their implementation in various programming models.

Received: 15 April 2010
Revised: 7 December 2010