Parallelizing Tableaux-Based Description Logic Reasoning

Thorsten Liebig and Felix Müller

Inst. of AI, University of Ulm, D-89069 Ulm, Germany
thorsten.liebig@uni-ulm.de,
felix.mueller@uni-ulm.de

Abstract. Practical scalability of Description Logic (DL) reasoning is an important premise for the adoption of OWL in a real-world setting. Many highly efficient optimizations for the DL tableau calculus have been invented over the last decades. None of them aimed at parallelizing the tableau algorithm itself. This paper describes our approach for concurrent computation of the nondeterministic choices inherent to the standard tableau procedure. We discuss how this interrelates with the well-known optimization techniques and present first promising performance results when benchmarking our prototypical reasoner *UUPR* (*Ulm University Parallel Reasoner*) with a selection of established DL systems.

1 Motivation

Tableaux-based algorithms have shown to be an adequate method in order to implement Description Logic (DL) reasoning services for many practical use-cases of moderate size. However, scalability of OWL reasoning is still an actual challenge of DL research [6]. Recent optimizations have shown significant increase in speed for answering queries with respect to large volumes of individual data under specific conditions. Unfortunately, almost all optimizations typically do come with some restriction in expressivity and end-users have to take care which approach to choose for a particular language fragment.

On the other hand, current processor families typically pool more than one processing unit on a single chip. Recent consumer desktops even have two quad-core processors on board. Today's reasoning engines unfortunately do not distribute their work load in such a setting. This is an unnecessary waste of computing power. Clearly, parallel computation can only reduce processing time by a factor which is limited by the available processing units but has the potential of being applicable without any restriction especially to the most "costly" cases.

This paper describes how to parallelize the well-known tableau algorithm as utilized sequentially within reasoning systems such as RacerPro, FaCT++, or Pellet. Our approach aims at parallelizing the tableau procedure itself rather than executing various instances of this procedure in parallel. The latter is a naive kind of parallelization whose synchronization may create some problems. This is because an optimized computation of the concept hierarchy does not consist of independent tasks as it will exploit previous subsumption results.

R. Meersman, Z. Tari, P. Herrero et al. (Eds.): OTM 2007 Ws, Part II, LNCS 4806, pp. 1135–1144, 2007.
© Springer-Verlag Berlin Heidelberg 2007

In contrast, parallelizing the nondeterministic choices within the standard DL tableau procedure has several advantages. First of all, nondeterminism is inherent to the tableau algorithm due to logical operators such as disjunction, at-most, or qualified cardinality restrictions. The generated alternatives from these expressions are completely independent of each other and can be computed concurrently. In case of a positive result the other sibling threads can be aborted. The parallel computation of nondeterministic alternatives also makes the algorithm less dependent on heuristics which otherwise have to choose the next alternative to process. For instance, a bad guess within a sequential algorithm inevitably will lead to a performance penalty. A parallel approach has the advantage of having better odds with respect to at least one good guess.

2 An Approach to Distributed DL Tableaux Proofs

Our approach aims at parallelizing the sequential algorithm proposed in [4] for $\mathcal{ALCNH}_{R^+}$ (also referred to as $\mathcal{SHN}$) ABoxes with GCIs.

Every standard reasoning task can be reduced to a corresponding ABox unsatisfiability problem. A tableau prover will then try to create a model for this ABox. This is done by building up a tree (the tableau) of generic individuals a_i (the nodes of the tableau) by applying tableaux expansion rules [1]. Tableaux expansion rules either decompose concept expressions, add new individuals or merge existing individuals.

2.1 Parallel Processing

The most obvious starting point for parallel evaluation are nondeterministic tableaux rules. Nondeterministic branching yields multiple alternatives, which can be seen as different possible ABoxes to continue reasoning with. In our setting, the following nondeterministic rules are covered:

The disjunction rule. If for an individual a the assertion $a : C \sqcup D$ is in the ABox $\mathcal{A}$, then there are two possible ABoxes to continue with, $\mathcal{A}' = \mathcal{A} \cup \{a : C\}$ or $\mathcal{A}'' = \mathcal{A} \cup \{a : D\}$.

The number restriction merge rule. If at an any point in the tableau there are m r successors of a in $\mathcal{A}$, $a : (\leq n\ r)$ is an assertion in $\mathcal{A}$ and $m > n$, the existing successors need to be merged to fulfill the restriction. An ABox $\mathcal{A}^i$ results for every possible combination.

As there are no dependencies between the alternatives, they can be evaluated within parallel threads.

To realize parallelism without recursively creating an overwhelming number of threads, we decided to implement a *work pool* design: A fixed number of threads is generated at the start of the tableau proof. This number typically will be equal to or less than the number of available processing units. These threads have synchronized read and write access to a common pool of jobs (i.e. the ABoxes to evaluate). In an initial step the tableaux root node (the original

ABox) is added to the pool. The executor starts the workers and one of them will fetch this job. In case of a nondeterministic rule application a worker will generate the necessary alternative ABoxes by creating copies of the preceeding ABox which are then submitted to the pool. These jobs will be processed by the next available workers. Figure 1 illustrates the resulting components within UUPR. The process is stopped when either

i) an ABox that represents a complete tableau is found, or
ii) no satisfiable alternative was found and there are no alternatives left to process.

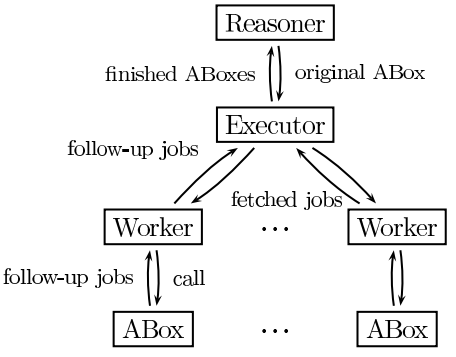


Fig. 1. Component interaction within work pool design of UUPR

An important decision in this design is the choice of the underlying pool data structure. The commonly used queue is unsuitable in this setting as it promotes a breadth-first style evaluation order. Thus, ABoxes which were created earlier (generated by fewer applications of nondeterministic rules) are preferred, and the discovery of complete ABoxes is delayed. The usage of a stack would not reliably lead to a depth-first oriented processing order either, because several threads can access the pool to put jobs into it.

We therefore chosed to use a priority queue in order to be able to explicitly influence the processing order. A simple heuristic to control the processing order:

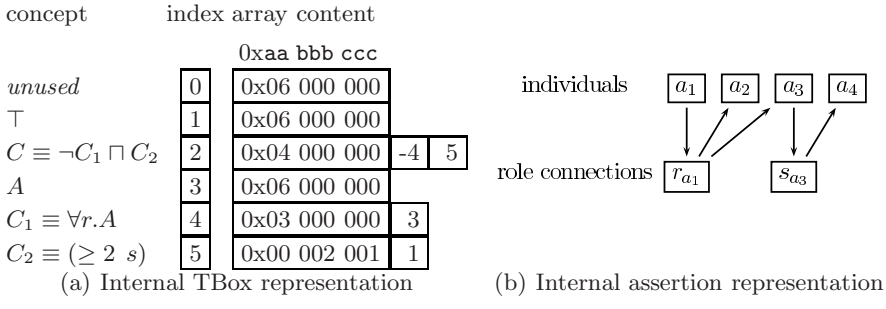
- The priority of the original ABox is set to 0.
- ABoxes generated from an ABox with priority n are given the priority $n+1$.

This allows for a controlled depth-first oriented processing order. More sophisticated heuristics or even some kind of A*-algorithm would also be possible. For example, FaCT++ also utilizes a priority queue for its ToDo list [9], weighting tableaux rules with different priorities. The difference is that FaCT++'s ToDo list contains all tableaux rules, while ours is restricted to nondeterministic rules (the other rules have a fixed order).

2.2 Data Representation

We also tried to design our internal data representation to be as efficient as possible. The main idea is to use integers to represent concepts and expressions (FaCT++ seems to use a similar encoding). Logical negation is realized through integer negation. The TBox is an array of concepts, with the most general concept $\top$ having index 1. When parsing a concept, numbers are recursively assigned to all subconcepts and subexpressions. Each indexed expression is represented as an integer array, where constructor, cardinalities and roles are all encoded into the first integer of this array. More precisely, the 32 bit of the first integer are split into three parts (`aabbbccc` in hexadecimal encoding). The first 8 bit encode the logical constructor. In case of a role constructor the next two chunks of 12 bit encode the cardinality value as well as the referenced role. A TBox, finally, is represented as an array of integer arrays. For example, Figure 2(a) shows the TBox containing the definition $C \equiv \neg\forall r.A \sqcap (\geq 2\ s)$ (note that the first integer for each indexed expression is shown in hexadecimal representation).

Assertions are collections of arrays for individuals and their role connections. I. e. for an individual a role-specific connection object is created in case of one (or more) fillers. In addition, each individual stores an associated concept assertion set which refers to indices of the TBox. An example assertion containing the assertions $(a_1, a_2) : r$, $(a_1, a_3) : r$ and $(a_3, a_4) : s$ is shown in Figure 2(b).



(a) Internal TBox representation (b) Internal assertion representation

Fig. 2. Example of UUPR's KB data structures

This compact representation guarantees low memory consumption and high processing speed. For instance, detecting a syntactic clash between two indexed expressions is reduced to a simple integer addition operation.

2.3 Optimizations

Today's state of the art reasoners achieve performance mainly through many highly efficient optimizations. Therefore, it is necessary to explore whether existing optimizations can be applied to our parallel architecture.

According to [9], DL tableau optimizations can be classified as follows:

– *Preprocessing and simplification*: As these optimizations are applied before the actual reasoning process is started, they are easy to combine with our approach. The most prominent optimizations of this kind is *GCI absorption*.

- *Optimizations in classification*: Current reasoners offer services to compute a taxonomy for a given ontology. Here, the number of subsumption tests can be reduced by exploiting implicitly computed subsumption relations between classes or by cheap syntactical (but incomplete) tests, such as pseudo model merging. For hierarchy computation, the applied subsumption algorithm is irrelevant, and thus a parallel subsumption algorithm can be used.
- *Optimizations in core satisfiability testing*: Optimizations that work directly in the reasoner core obviously are those which may interact with a parallel reasoner architecture. However, many of them are very important even in prototypical reasoners, since a naive implementation will often lead to practical nontermination even for small knowledge bases.

As a proof of concept we added the following known (mostly core reasoner) optimizations to our UUPR implementation:

Naming and Lazy Unfolding. These two techniques are fundamental to DL reasoning engines and integrated deeply in our reasoner core. Naming is done be recursively assigning names to all occurring subconcepts and is reflected by the internal data representation described in Figure 2(a). Lazy unfolding means that these names are only expanded when needed.

Lexical Normalization. This is a preprocessing optimization and aims to normalize input data, such that inconsistencies are detected as early as possible. Lexical normalization includes a number of syntactical simplification rules.

Semantic Branching. Semantic Branching is a technique similar to the DPLL procedure used in propositional satisfiability testing. It influences the way in which alternatives are generated during reasoning. The main idea is to avoid having to solve the same sub-problems in multiple alternatives by explicitly making them distinct: For instance, for $A \sqcup B$, A and $\neg A$ (respectively) are added to the alternatives.

Simplification. Simplification tries to reduce the amount of nondeterminism by avoiding unnecessary branching. It is a technique similar to boolean constraint propagation (BCP). For example, when $\neg A \sqcap (A \sqcup B)$ is contained in a tableau node, no branching is necessary and B can be added to the node.

Caching. This is applied to all calculated results of subproblems encountered during the reasoning process.

From the above optimizations, only caching leads to additional synchronization overhead, as cache accesses are mutually exclusive. All other optimizations were integrated without any special adaption into our reasoner core.

2.4 Implementation

Our parallel ABox reasoner is implemented in C++ as a shared memory program using the boost.Threads library[1]. More precisely, UUPR is developed for the SMP (symmetric multi processor) architecture, where all processors have access to one main memory. Unlike in Java or Lisp, programmers using C++ can

[1] http://www.boost.org/

influence the way memory allocation is managed. The standard template library (STL) container classes normally use the `std::allocator`. However, using the latter, no parallel speed-up could be achieved. In fact, using more processors only resulted in decreasing program performance as shown in Figure 3.

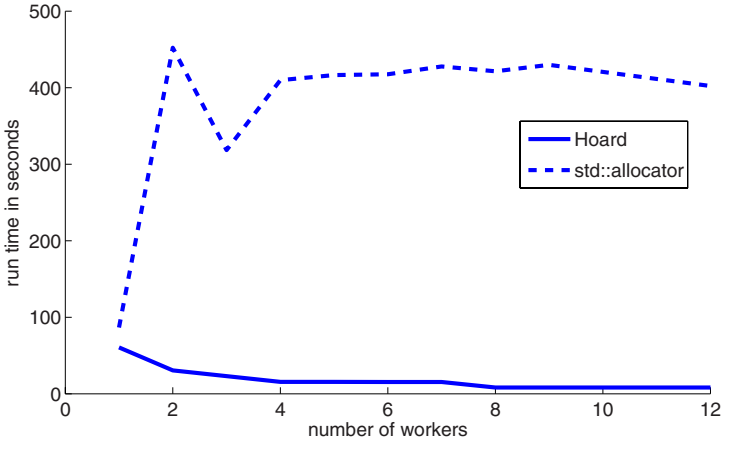


Fig. 3. Impact of memory manager

Fortunately, the STL classes can be parameterized to use a different memory allocator. It turned out that the superior heap organization utilized in memory allocators such as Hoard[2] [2] is an essential premise for any performance gain in a parallel shared memory environment. Consequently, UUPR can be compiled with one of two memory managers specifically developed for use in parallel programs, the Hoard library as well as Intel's thread building blocks (TBB)[3]. The number of parallel workers can be specified as a parameter at run time.

3 Experimental Results

Our performance tests were run on the following platforms:

- A Sun compute server with 12 UltraSPARC IV+ dual core processors, running at 1.8 GHz each, and 96 GB of main memory. The processor load was about 50%, so effectively, at best we had about 12 processors during testing.
- An ubuntu Linux system with two AMD Opteron dual core processors, which run at 2.2 GHz and 16 GB of main memory.
- As a standard desktop computers we used a 2.4 GHz dual core AMD desktop computer with 1 GB main memory running Suse Linux.

[2] http://www.hoard.org/
[3] http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm

– For comparison, a 3 GHz single core computer with 1 GB of main memory is also included. We also made spot tests on a MacBook Core 2 Duo, which yielded results similar to the dual core desktop machine.

Three test cases were selected for evaluation:

Filler merging. Test case 1 is taken from [7] (2b). Checking satisfiability of $X2$ will create a lot of role successors, which then must be merged due to a maximum cardinality restriction: $X2 \equiv \exists r.C1 \sqcap \ldots \sqcap \exists r.C15 \sqcap (\leq 2\ r)$. Three of the Ci are mutually disjoint, so that all possible combinations need to be examined. This leads to a lot of nondeterminism and many small ABoxes have to be checked, making the synchronized pool access a possible performance limitation. Results are shown in Figure 4.

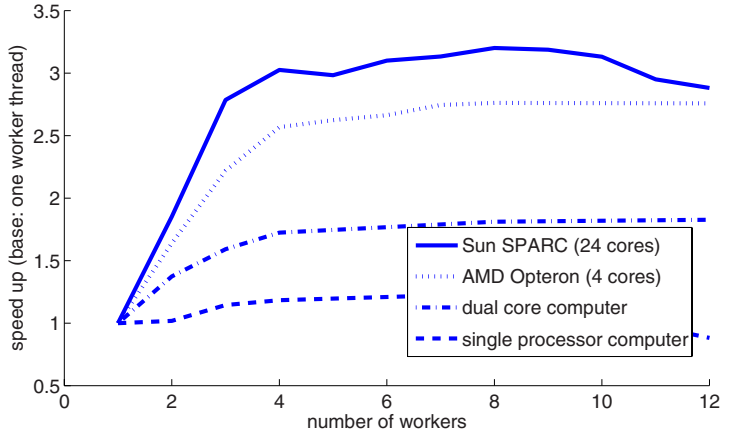


Fig. 4. Speed-up for test case 1 started with different numbers of workers

Disjunction. Test case 2 is an extended version of case 28 from [7]. It is designed to be a costly satisfiability test of a concept A without any nondeterminism. Here we check for satisfiability of a concept C, defined as a disjunction of eight concepts similar to A: $C \equiv A_1 \sqcup \ldots \sqcup A_8$. Since semantic branching was disabled here, the result is an equal distribution of 8 costly tasks on workers with low synchronization overhead. Therefore, Figure 5 shows the effect of a step-wise speed-up whenever the number of workers is a divider of 8.

Realistic ontology. To determine performance on a more realistic ontology and to demonstrate the applicability of our approach to all of $\mathcal{SHN}$, we took the example ontology given in [4]. It models a family ontology, using TBox and ABox knowledge as well as GCIs. We added a subtle contradiction and performed an ABox satisfiability check. Results are given in Figure 6.

A considerable speed-up can be observed in all cases. Except for the Sun the performance increase is almost linear up to number of available processing units. The performance decline for the Sun platform presumably has two

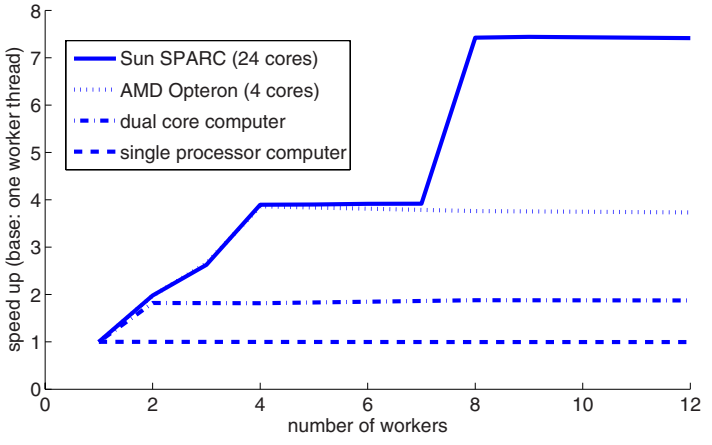


Fig. 5. Speed-up for test case 2 started with different numbers of workers

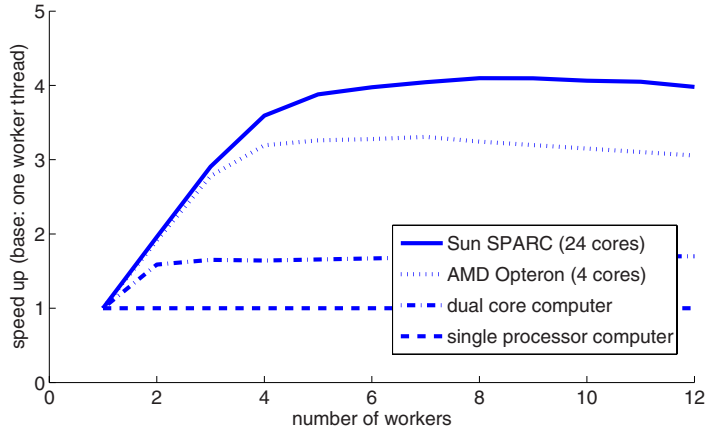


Fig. 6. Speed-up for family ontology started with different numbers of workers

reasons. First, the Sun was heavily used by many other users during testing. Second, a lot of nondeterminism occurs in the cases 1 and 3. This results in many memory allocation operations (when ABoxes are created) and requires a lot of synchronization (for work pool access). This extra effort is proportional to the number of concurrently executed worker threads exclusively running on their own processing core. Whether memory bandwidth or thread synchronization is the limiting factor here is subject to further investigations.

Table 1 shows a comparison of UUPR's respective best performance with a selection of other reasoners. For test case 3 we reproducibly got a segmentation fault for UUPR when deactivating the optimizations of sec. 2.3 (except naming and lazy unfolding) on one of our test environments (Sun), while there (again

Table 1. Comparison with other systems on Sun

System	test case 1	test case 2	test case 3
KAON2	*memout*	*timeout*	2.490s
Pellet 1.4	*timeout*	144.921s	2.375s
UUPR with optimizations	56.834s	8.152s	13.122s
UUPR without optimizations	60.401s	8.794s	*failed*

reproducibly) was a time out (10 min) on the other platforms. A plausible reason for this could not be determined. The most noticable difference was the different memory manager (Hoard on Sun, Intel TBB for the others).

4 Related Work

As far as we know there is no approach aiming at parallelizing the tableau calculus particularly for DL reasoning. However, there are a couple of tableau-based theorem provers capable of parallelizing obviously independent parts of the search tree, such as or-parallelism (e. g. Meteor, PartabX, Parthenon, SETHEO) [8]. Current work seems to be only in the niche of temporal reasoning and is at best expected to become an integral part of provers in a couple of years [10].

Older DL-related work deals with parallel processing of the structural algorithms utilized by the FLEX system. According to [3], the most promising processing phases for parallelization is the execution of propagation rules for ABox realization as opposed to the structural algorithm for TBox reasoning such as normalization or comparison. They argue that within the setting of a MIMD (Multiple Instruction, Multiple Data) system the basic operations during structural classification are too fine-grained for efficient parallelization.

5 Outlook

Our parallel reasoner has shown encouraging first results. The approach can be combined with well-known optimizations and bears no restriction which would prevent it from being extended to more expressive language fragments (even $\mathcal{SROIQ}$). At the same time there are many extensive optimizations conceivable.

An obvious way to further increase the performance of our approach is to employ techniques to reduce the amount of synchronized work pool access to a ratio compatible with our design. This can, for example, be done by cutting off parallel computation above a given problem size or branching factor and by implementing known optimizations such as GCI absorption. Another optimization is also missing: dependency directed backtracking. The latter is more difficult to implement because it requires to keep track of references to previous nodes of the tableau, which are currently missing within our design.

A particular parallel optimization refers to cascading work pools and/or caches, so that only a small number of threads share one work pool or cache. This

would drastically reduce synchronization efforts, especially with larger numbers of worker threads. In addition, it would potentially allow to execute the threads of one work pool on a different machine. Although not an issue within our evaluation, memory could be saved by replacing ABox cloning by ABox structure sharing across threads in case of nondeterministic alternatives.

Extensions to the parallel evaluation itself are another option. For instance, qualified cardinality restrictions, as offered by OWL 1.1, add nondeterminism due to their choose-rule. Another idea is the parallel evaluation of conjunctions. In principle, the conjuncts C and D of a conjunction $C \sqcap D$ can be evaluated in parallel. The problem here are mutual dependencies between the conjuncts. For example, in the conjunction $\forall r.A \sqcap \exists r.\neg A$ the clash would not be detected if the conjuncts were processed in parallel. Therfore, a dependency test is needed to determine whether parallel evaluation is possible, that is, to check whether two conjuncts do interact in some way. A technique similar to pseudo-model merging [5] of sub-expressions (not only root nodes) could be used to achieve this.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York (2003)
2. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A scalable memory allocator for multithreaded applications. In: International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, MA, pp. 117–128 (November 2000)
3. Bergmann, F.W., Quantz, J.J.: Parallelizing Description Logics. In: Wachsmuth, I., Brauer, W., Rollinger, C.-R. (eds.) KI-95: Advances in Artificial Intelligence. LNCS, vol. 981, pp. 137–148. Springer, Heidelberg (1995)
4. Haarslev, V., Möller, R.: Expressive ABox Reasoning with Number Restrictions, Role Hierarchies, and Transitively Closed Roles. In: Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2000), pp. 273–284 (2000)
5. Haarslev, V., Möller, R., Turhan, A.Y.: Exploiting Pseudo Models for TBox and ABox Reasoning in Expressive Description Logics. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 29–44. Springer, Heidelberg (2001)
6. Horrocks, I.: Applications of description logics: State of the art and research challenges. In: Dau, F., Mugnier, M.-L., Stumme, G. (eds.) ICCS 2005. LNCS (LNAI), vol. 3596, pp. 78–90. Springer, Heidelberg (2005)
7. Liebig, T.: Reasoning with OWL – system support and insights –. Technical Report TR-2006-04, Ulm University, Ulm, Germany (September 2006)
8. Schumann, J.: Tableau-Based Theorem Provers: Systems and Implementations. Journal of Automated Reasoning 13, 409–421 (1994)
9. Tsarkov, D., Horrocks, I., Patel-Schneider, P.F.: Optimising Terminological Reasoning for Expressive Description Logics. Journal of Automated Reasoning 39(3), 277–316 (2007)
10. Voronkov, A.: Automated Reasoning: Past Story and New Trends. In: Proc. of the Int. Joint Conf. on AI (IJCAI-2003, pp. 1607–1612 (2003)