Implementation and Evaluation of a Tableau Algorithm for the Guarded Fragment

Jan Hladik

Theoretical Computer Science
RWTH Aachen

Abstract. In this paper we present SAGA, an implementation of a tableau-based **S**atisfiability **A**lgorithm for the **Gu**arded Fragment ($\mathcal{GF}$). Satisfiability for $\mathcal{GF}$ with finite signature is EXPTIME-complete and therefore intractable in the worst case, but existing tableau-based systems for EXPTIME-complete description and modal logics perform reasonably well for "realistic" knowledge bases. We implemented and evaluated several optimizations used in description logic systems, and our results show that, with an efficient combination, SAGA can compete with existing highly optimized systems for description logics.

1 Preliminaries

The Guarded Fragment of first order predicate logic ($\mathcal{GF}$) [1] restricts the appearance of quantifiers to formulae of the kind

$$\forall \boldsymbol{x}(G(\boldsymbol{x},\boldsymbol{y}) \rightarrow \varphi(\boldsymbol{x},\boldsymbol{y}))$$
$$\exists \boldsymbol{x}(G(\boldsymbol{x},\boldsymbol{y}) \wedge \varphi(\boldsymbol{x},\boldsymbol{y})) \ ,$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables, and $G(\boldsymbol{x},\boldsymbol{y})$ is an atom (the *guard* of the formula). This fragment has many desirable properties: satisfiability is decidable [1] in 2-EXPTIME, which is reduced to EXPTIME if the arity of the relations is bounded [11]. It has the finite model property [1] and a tree model property for a special kind of tree [11].

$\mathcal{GF}$ can be regarded as a generalization of modal or description logics to n-ary relations (roles). The more expressive such logics have a comparably high worst-case complexity, e.g. PDL [5] and $\mathcal{SHIQ}$ [21] are EXPTIME-complete [27, 20]. However, with systems deciding one of these logics using an optimized tableau algorithm, e.g. DLP [25], FaCT [18], and RACER [13], satisfiability becomes tractable for a variety of realistic knowledge bases [18, 19, 14]. This suggests that a tableau algorithm for $\mathcal{GF}$-satisfiability might lead to an implementation that does not consume exponential time for "realistic" formulae.

$\mathcal{GF}$ is also decidable by resolution [9]. To the best of our knowledge, the efficiency of this approach for realistic formulae has not yet been analyzed.

2 A Tableau Algorithm for $\mathcal{GF}$

In [16], a tableau algorithm is presented for the Clique Guarded Fragment ($\mathcal{CGF}$) [10], a generalization of $\mathcal{GF}$. Its worst-case complexity is 2-NEXPTIME (NEXPTIME for signatures with bounded arity), and therefore higher than that of $\mathcal{GF}$ itself, but it allows for many of the optimizations known from description logics and therefore promises to lead to an efficient implementation. SAGA only implements those parts of the algorithm that are relevant for $\mathcal{GF}$.

Due to space limitations, we cannot present the entire algorithm here. Instead, we will only describe how it differs from "standard" tableau algorithms for description logics (for an introduction and examples, see e.g. [2]). A node in a $\mathcal{GF}$ tableau does not stand for a single constant (individual), but for a set of constants that appear together in a guard atom. Therefore, a node n is labeled with a set $C(n)$ of constants and a set $\Delta(n)$ of formulae which contain only constants in $C(n)$.

A tree model for a $\mathcal{GF}$ formula need not be finite. For that reason, termination of the tableau generation is ensured by *blocking*, i.e. no successors are created for a node n if there exists an ancestor m of n which contains the same formulae as n (modulo a mapping π between the constants of n and m). For this purpose, n is labeled upon creation with a natural number $N(n)$ which is larger than $N(l)$ for every node l that was created previously.

Formally, a node n is *directly blocked* if there exists a node m such that $N(m) < N(n)$, m is not blocked, and there exists an injective mapping $\pi\colon C(n) \to C(m)$ such that for all constants $c \in C(n) \cap C(m)$, $\pi(c) = c$ and for the extension of π to formulae, $\pi(\Delta(n)) = \Delta(m) \mid_{\pi(C(n))}$ holds. A node n is *blocked* if it is directly blocked or if its predecessor is blocked. This notion of blocking is not equivalent to *subset blocking*, where a node n is blocked by a node m if the label of n is a subset of the label of m: for $\mathcal{GF}$, the image of π need only be a subset of $C(m)$, but restricted to these constants, the labels of u and v have to be equal (modulo π).

3 Optimization

In Section 2, we mentioned that optimizations are necessary to obtain practical decidability for description logics. In the following, the optimizations included in SAGA are described.

Syntactic Simplification. Before SAGA tries to construct a tableau for the input formula φ it simplifies the syntactic structure of φ to speed up the tableau generation process: tautologies and contradictions are made explicit, a normal form is used which supports their detection, and the variables contained in a formula φ are normalized when φ is added to a node. Details can be found in [17] or [18].

Semantic Branching. The naive method to satisfy a disjunction $\varphi \vee \chi$ is to add φ first and, if this causes a clash, add χ afterwards (*syntactic branching*). This is rather inefficient because resources have been spent to find out that φ is unsatisfiable in the current tree, but this information is not used any more. *Semantic branching* [7] adds $\neg\varphi \wedge \chi$ to the tree if φ leads to a clash. This makes explicit the information that φ is unsatisfiable and prunes the search space because a tree in which φ is satisfiable is never tested again.

Backjumping. After a clash, naive backtracking returns to the most recent branching point (BP). Dependency directed backtracking (*backjumping*) [3] instead returns to the most recent BP *one of the clashing formulae depends on.* Thus, the intermediate BPs, which did not have any influence on the clash, are skipped. To make backjumping possible, every formula φ in a node n is labeled with a *dependency set* $D(\varphi, n)$, the set of branching points the presence of φ in n depends on. After a clash between φ and χ in n, the most recent BP b in $D(\varphi, n) \cup D(\chi, n)$ is determined and the backjump to b is performed.

Boolean Constraint Propagation. Before choosing a disjunct ψ from a disjunction $\varphi \vee \chi$ in a node n and performing a branch for ψ, every disjunct is tested for being *closed*: a disjunct is closed if its negation is already contained in n; otherwise, it is *open.* If ψ is closed, it is removed from the disjunction (because adding ψ would lead to an immediate clash), and only the remaining disjuncts are considered for branching (*boolean constraint propagation* [6]). In particular, if there is only one open disjunct, it is added deterministically to n, and the branch is avoided.

4 Heuristics

There are two kinds of non-determinism involved in the $\mathcal{GF}$-algorithm: the decision on which one of several available formulae to process first is *don't-care* non-deterministic, i.e. every choice will lead to a correct behaviour of the algorithm, whereas the decision on which disjunct of a disjunction to add to the corresponding node (*branching*) is *don't-know* non-deterministic, i.e. only certain choices will lead to the discovery of a tableau. In both cases, the heuristic used to make the decision obviously has a significant influence on performance.

Branching. In SAGA, three different heuristics for choosing the disjunct for the next branch are implemented. Each one tries to improve the efficiency of one of the other optimizations.

MOMS The heuristic "**M**aximum **O**ccurrence in disjunctions of **M**inimum **S**ize" [6] considers *all* disjunctions of minimum size in the corresponding node and counts the positive and negated appearances of the disjuncts. The disjunct φ to branch on is the one with the largest counter. If the counter for φ is larger than the one for $\neg\varphi$, $\neg\varphi$ is tested first. If this leads to a clash, φ is tested afterwards. (MOMS therefore implicitly requires semantic branching.) The goal is to optimize BCP by increasing the number of closed disjuncts and reaching deterministic expansion as soon as possible. One disadvantage of MOMS lies in the fact that it tries the more constrained alternative first, i.e. the alternative that is more likely to fail [17]. Furthermore, it was observed that MOMS interacts adversely with backjumping [18].

iMOMS Inverted MOMS [17] tries to avoid the disadvantage of being likely to fail with the first alternative by testing φ and $\neg\varphi$ in the opposite order. Thus, it first chooses the disjunct which satisfies most of the (smallest) disjunctions.

Maximize-jump This heuristic was developed for the FaCT system [19]. From all disjunctions of a node, it selects the one for which the maximum element in the dependency set is minimal, i.e. the one leading to the furthest backjump. In order to find the first disjunct to try, FaCT uses a MOMS-style heuristic. In SAGA, the syntactically shortest disjunct is selected because a short formula probably can be tested faster than a long one which is likely to contain existential or universal formulae. Since this approach does not rely on counts of disjuncts like MOMS, we also expect to see the differences in efficiency more clearly.

Choosing the Next Formula. There are two kinds of formulae which are significantly more expensive to process than the other ones: disjunctions require branching and backtracking, which includes creating backups of nodes and restoring them after a clash, and for existential formulae in a node n, each predecessor m_i of n has to be compared with all nodes l_j with $N(l_j) < N(m_i)$. For the blocking test itself, all mappings from $C(n_i)$ to $C(l_j)$ have to be tested (in the worst case). Therefore, either disjunctions or existential formulae should be processed last. These two heuristics are implemented in SAGA.

Blocking The blocking condition for a node n as defined in Section 2 requires $\pi(\Delta(n))$ to be equal to a *restriction* $\Delta(m)\mid_{\pi(C(n))}$ for a predecessor node m. The algorithm also works for an alternative definition of blocking where the same number of constants is required for n and m, i.e. equality of $\Delta(n)$ and $\Delta(m)$ modulo π. This may lead to postponing blocking because the blocking test only succeeds after the creation of some additional nodes, but the test itself becomes significantly more efficient: if the number of constants, atoms, universal formulae etc. is not identical for m and n, it can be aborted immediately without generating a mapping π. In the following, we will refer to the different blocking conditions as *subset-equality blocking* and *equality blocking* respectively.

5 Comparison of Heuristics

In this section, we present an analysis of the efficiency of the heuristics and optimizations described in the previous sections. We used two sets from the "Tableaux 2000 Non-Classical Systems Comparison" (TANCS-2000) [24] benchmark suite and some $\mathcal{GF}$ formulae to see how the heuristics behave for formulae of different complexity.

QBF-INV The "quantified boolean formulae" benchmark consists of sets of 8 random QBF formulae which satisfy given parameters. These formulae are translated into the logic $\mathbf{K}^-$ ($\mathbf{K}$ with inverse modality). For this comparison, we used the sets "p-qbf-inv-cnfSSS-K4-Cc-V4-D4" with $c \in \{10, 20, 30, 40, 50\}$.

PSAT-INV The random generated "periodic satisfiability" formulae are translated into the logic $\mathbf{K}^-$ with global axioms. We used the sets "p-psat-inv-cnf-K4-Cc-V4-D4" with $c \in \{20, 30, 40, 50\}$.

GFB The QBF and PSAT formulae do not allow us to evaluate the different blocking conditions (equality or subset-equality blocking, cf. Section 4) because every node in a tree constructed for these formulae contains exactly two constants. Therefore, we generated some (simple) "$\mathcal{GF}$ Benchmark" (GFB) formulae. Each set consists of 8 formulae and is characterized by the width w and depth d of the formula and the maximum arity r of the relations.

The QBF benchmark does not require blocking because termination of the algorithm is ensured by the properties of the $\mathbf{K}^-$ logic: the complexity of formulae continuously decreases from predecessor to successor nodes. This property makes it possible to regard the blocking test as another heuristic for these logics and evaluate its efficiency by turning it on or off.

To evaluate the heuristics by themselves as well as their interactions, we ran every benchmark with every possible combination of heuristics. The figures in the following sections show how many formulae could be solved for the corresponding combination. The benchmarks were run on the following system: hardware: Pentium-III (733 MHz), 384 MB RAM, 512 MB swap space; software: Linux (Kernel 2.2), Allegro Common Lisp 6.0. Timeout: 600 sec (TANCS); 100 sec (GFB).

QBF Results (Figure 1). Surprisingly, blocking is the most efficient heuristic. With blocking enabled, up to 26 formulae can be solved, compared to at most 4 without blocking. Although the blocking test is very expensive in the worst case, it is obviously far more efficient than the expansion of the nodes that can be blocked.

Semantic branching and backjumping also provide a significant speedup. While backjumping delivers a rather constant improvement independent of the other optimizations, semantic branching works particularly well with efficient combinations. iMOMS is slightly worse than maximize-jump, and MOMS is far worse than the other branching heuristics. This is true even when backjumping is disabled, i.e. when maximize-jump effectively chooses a random disjunct (because there is no backjump to maximize). Processing $\exists$- or $\vee$-formulae last does not have a significant influence, and syntactic simplification has none at all (it is therefore not recorded in the figures).

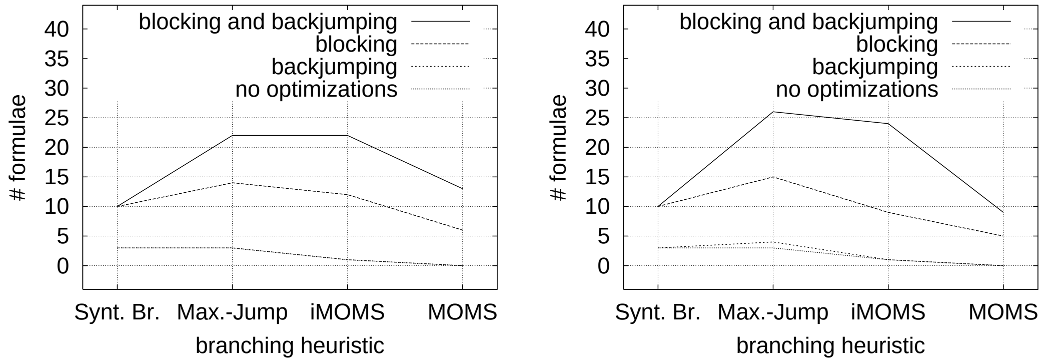

Fig. 1: Heuristics Comparison for QBF with "∨ Last" (Left) and "∃ Last" (Right)

PSAT Results (Figure 2). Semantic branching is by far the most important heuristic, and backjumping is also very efficient. Furthermore, we can observe a significant speedup if ∃-formulae are processed last. iMOMS is slightly better than maximize-jump, but the difference is irrelevant for efficient combinations of the other heuristics. The same holds for syntactic simplification: if we have an efficient combination of heuristics, disabling syntactic simplification does not significantly slow down the system. This indicates that, in the presence of semantic branching and backjumping, a complicated syntax of a formula does not affect the overall efficiency.

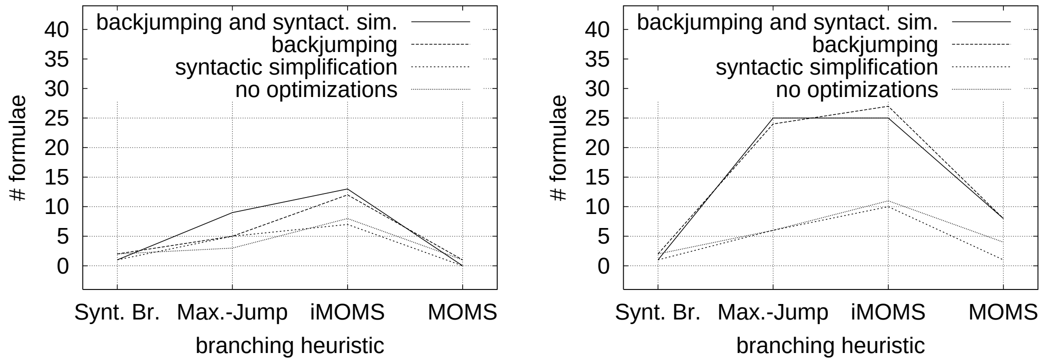

Fig. 2: Heuristics Comparison for PSAT with "∨ Last" (Left) and "∃ Last" (Right)

GFB Results. The results for QBF indicate that blocking, although expensive, results in a speedup. This raises the question if searching for more blocking situations by using subset-equality blocking instead of equality blocking might lead to a further speedup. The tests show that this is is not the case: subset-equality blocking leads to a higher average time and to fewer solvable formulae. Syntactic simplification has more influence for GFB than for the other benchmarks, which is probably caused by the simple and random structure of the formulae. The remaining heuristics behave similarly (therefore we do not include a figure).

6 Comparison with Other Systems

In this section, we compare the results of the TANCS-2000 competition with those we obtained with SAGA. This allows us to examine how well SAGA scales compared to other systems, i.e. how fast it can solve formulae from $\mathcal{GF}$ and less expressive logics. Since most of the TANCS systems used the formulae without inverse roles for benchmarking, the results presented in Figure 3 were produced with the "PSAT" rather than with the "PSAT-INV" formulae.

RACE [12, 15] is a TBox and ABox reasoner for the description logic $\mathcal{ALCNH}_{R^+}$. DLP [26, 25] is a DL system for several expressive description logics, including **K** and PDL. In addition to the optimizations used in SAGA, these two systems perform *caching*, i.e. they remember the satisfiability status of a node to avoid the expansion of another node with the same label. The third competitor SPASS [28, 29] is a resolution based system testing satisfiability of first order formulae. It also contains several optimizations, some of which resemble those mentioned in Section 3 (e.g. "branch condensing" corresponds to backjumping [22]). MSPASS [23] is a SPASS module translating formulae from the syntax of of modal and description logics to first order logic so they can be tested with SPASS.

For this comparison, we used the formulae "p-psat-cnf-K4-Cc-Vv-Dd" with $c \in \{20, 30, 40, 50\}, v \in \{4, 8\}, d \in \{1, 2\}$ and the GFB formulae. The figures show for every system how many formulae of each set could be solved. The SAGA benchmark was run on the following system: hardware: Pentium-III (1 GHz), 512 MB RAM, 1 GB swap space; software: Linux (Kernel 2.2), Allegro Common Lisp 6.0. Timeout: 1000 sec.

PSAT Results (Figure 3). Although the results presented in this paragraph were produced on different hardware with different timeouts and are not directly comparable, the differences between the systems are relatively small. With depth 1, the formulae were easy for all of the systems. For the harder formulae with depth 2, the differences become visible and it turns out that SAGA is slightly slower than RACE and slightly faster than DLP and MSPASS.

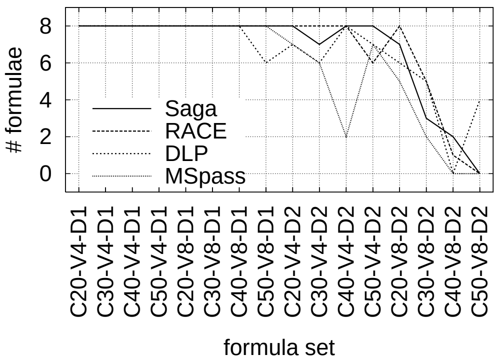

Fig. 3: System Comparison for PSAT

GFB Results (Figure 4). The formula sets are characterized by their w, d and r values (cf. Section 5). Among the comparison systems, SPASS is the only one that can decide $\mathcal{GF}$ formulae, but its characteristics are quite different from SAGA's. Therefore, it is no surprise that the results are also different: while SAGA handles a large depth (d16) of a formula well, it has problems with relations of a higher arity (r4). SPASS shows the opposite behaviour. The sum of decidable formulae is similar.

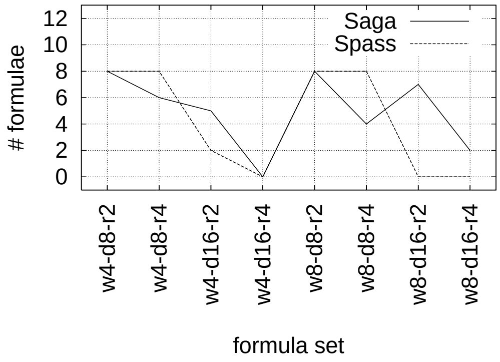

Fig. 4: System Comparison for GFB

7 Conclusion

In this paper, we presented SAGA, an implementation of a $\mathcal{GF}$ tableau algorithm. It turned out that in spite of its high worst-case complexity, it performs well for existing benchmark formulae if an efficient combination of optimizations is used: semantic branching and backjumping are particularly useful. The blocking test speeds up the program even when blocking is not required to ensure termination.

Among the branching heuristics, maximizing the backjump or satisfying many disjunctions at once is most efficient. Whether disjunctions or existential formulae should be processed last depends on the logic of the formula to test. In comparison to other systems, SAGA is slightly faster than DLP and SPASS/MSPASS and slightly slower than RACE.

Acknowledgements

The author of this paper is supported by the DFG, Project No. GR 1324/3-3.

References

[1] H. Andréka, J. van Benthem, and I. Németi. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[2] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69, 2001.

[3] A. B. Baker. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results.* PhD thesis, University of Oregon, 1995.

[4] R. Dyckhoff, editor. *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX 2000)*, volume 1847 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

[5] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Science*, 18:194–211, 1979.

[6] J. W. Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, 1995.

[7] J. W. Freeman. Hard random 3-SAT problems and the Davis-Putnam procedure. *Artificial Intelligence*, 81:183–198, 1996.

[8] H. Ganzinger, editor. *Proceedings of the 16th International Conference on Automated Deduction (CADE-99)*, volume 1632 of *Lecture Notes In Artificial Intelligence*, Berlin, 1999. Springer-Verlag.

[9] H. Ganzinger and H. de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, Trento, Italy, 1999. IEEE Computer Society Press.

[10] E. Grädel. Decision procedures for guarded logics. In Ganzinger [8].

[11] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999.

[12] V. Haarslev and R. Möller. Consistency testing: The RACE experience. In Dyckhoff [4].

[13] V. Haarslev and R. Möller. Description of the RACER system and ist applications. In D.L. McGuinness, P.F. Patel-Schneider, C. Goble, and R. Möller, editors, *Proceedings of the 2001 International Workshop on Description Logics (DL2001)*, volume 49, Stanford, CA, USA, 2001. CEUR.

[14] V. Haarslev and R. Möller. High performance reasoning with very large knowledge bases: A practical case study. In Bernhard Nebel, editor, *Proceedings of the Seventeenth International Joint Conference on Aritificial Intelligence*. Morgan Kaufman Publishers, San Francisco, USA, 2001.

[15] V. Haarslev, R. Möller, and A.-Y. Turhan. RACE user's guide and reference manual version 1.1. Technical Report FBI-HH-M-289/99, University of Hamburg, CS department, 1999.

[16] C. Hirsch and S. Tobies. A tableau algorithm for the clique guarded fragment. In F. Wolter, H. Wansing, M. de Rijke, and M. Zakharyaschev, editors, *Advances in Modal Logics*, volume 3, Stanford, 2001. CSLI Publications.

[17] J. Hladik. Implementing the n-ary description logic GF1-. In F. Baader and U. Sattler, editors, *Proceedings of the International Workshop on Description Logics*, Aachen, Germany, 2000. CEUR.

[18] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[19] I. Horrocks and P. F. Patel-Schneider. Optimising description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[20] I. Horrocks and U. Sattler. A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410, 1999.

[21] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes In Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[22] U. Hustadt and R. A. Schmidt. MSPASS: Modal reasoning by translation and first-order resolution. In Dyckhoff [4].

[23] U. Hustadt, R. A. Schmidt, and C. Weidenbach. MSPASS: Subsumption testing with SPASS. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics*, Linköping, Sweden, 1999. CEUR.

[24] F. Massacci and F. M. Donini. Design and results of TANCS-00. In Dyckhoff [4].

[25] P. F. Patel-Schneider. DLP system description. In E. Franconi, G. De Giacomo, R. M. MacGregor, W. Nutt, and C. A. Welty, editors, *Proceedings of the International Workshop on Description Logics*, Povo - Trento, Italy, 1998. CEUR.

[26] P. F. Patel-Schneider. TANCS-2000 results for DLP. In Dyckhoff [4].

[27] V. R. Pratt. Models of program logics. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, 1979.

[28] C. Weidenbach. SPASS: Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 27. Elsevier, 1999.

[29] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In Ganzinger [8].