Implementing the n-ary Description Logic $\mathcal{GF}1^-$

Jan Hladik
LuFG Theoretical Computer Science, RWTH Aachen, Germany
email: jan@cantor.informatik.rwth-aachen.de

1 Introduction

$\mathcal{GF}1^-$ is a decidable description logic allowing for n-ary relations. It was introduced in [LST99] along with a tableau algorithm deciding $\mathcal{GF}1^-$-satisfiability in PSPACE. In this paper, the implementation of this algorithm, the modifications and optimizations used, and some empirical results are described.

2 Preliminaries

In the following, upper case letters stand for relations, lower case letters $c, d, \dots$ for constants, and $v, w, x, \dots$ for variables. $\overline{x}$ denotes a non-empty variable vector, $\varphi, \chi, \dots$ closed formulae, $\varphi(\overline{x})$ a formula with free variables $\overline{x}$. $\varphi(\frac{x}{c})$ means the formula φ with all occurrences of x replaced with c, and $\varphi(\frac{\overline{x}}{\overline{c}})$ is defined accordingly.

$\mathcal{GF}1^-$ syntax and semantics The Guarded Fragment 1 as described in [ANvB98] restricts First Order Predicate Logic by allowing quantifiers to appear only in formulae of the kind:

$$\forall \overline{x}(G(\overline{x}, \overline{y}) \rightarrow \varphi(\overline{x})) \qquad \text{or} \qquad \exists \overline{x}(G(\overline{x}, \overline{y}) \wedge \varphi(\overline{x}))$$

where G must be an atom and is called the *guard* of the formula.

$\mathcal{GF}1^-$ imposes the further restriction that for every predicate P with arity n there is a "grouping" (i, j) of its parameters such that $n = i + j$ (written as $P^{(i,j)}$), $\overline{x}$ is of length i or j, $\overline{y}$ is of the remaining length, and $\overline{x}$ and $\overline{y}$ do not share any variables, i.e.

$$\mathcal{Q}\overline{x}(G^{(i,j)}(\overline{x}, \overline{y})) \circ \varphi(\overline{x}) \qquad \text{or} \qquad \mathcal{Q}\overline{x}(G^{(j,i)}(\overline{y}, \overline{x})) \circ \varphi(\overline{x})$$

where $\text{length}(\overline{x}) = i$, $\mathcal{Q} \in \{\forall, \exists\}$, and $\circ$ is either $\wedge$ or $\rightarrow$, depending on the quantifier.

The semantics for $\mathcal{GF}1^-$ is defined just like for standard first order logic.

A tableau-based algorithm for $\mathcal{GF}1^-$ The tableau algorithm deciding $\mathcal{GF}1^-$-satisfiability [LST99] is similar to the $\mathcal{ALCI}$ algorithm [Spa93]. It first replaces all free variables in a given formula with constants. These constants are added to the root node of the model. Then, de Morgan's laws are used to transform the formula into *Negation Normal Form* (NNF) by pushing negation inwards such that it only occurs in front of predicates. This way, we only need rules for four kinds of formulae:

- $\varphi \wedge \chi$: φ and χ are added to the node.
- $\varphi \vee \chi$: a nondeterministic choice is made between adding φ or χ.
- $\exists \overline{x}(G(\overline{d}, \overline{x}) \wedge \varphi(\overline{x}))$, where $\overline{d}$ consists of constants of the current node: the variables in $\overline{x}$ are replaced with new constants $\overline{c}$. A new node is created with constants $\overline{c}$ and the formula $\varphi(\frac{\overline{x}}{\overline{c}})$. The edge connecting the nodes is labeled with $G(\overline{d}, \overline{c})$.
- $\forall \overline{x}(G(\overline{d}, \overline{x}) \rightarrow \varphi(\overline{x}))$, $\overline{d}$ as above: all formulae of the current node and the guards connecting it with its father and sons are checked for matching $G(\overline{d}, \overline{x})$. For every match $G(\overline{d}, \frac{\overline{x}}{\overline{c}})$, $\varphi(\frac{\overline{x}}{\overline{c}})$ is added to the corresponding node.

Rules are only *applicable* if the formulae they would add are not yet a member of the corresponding node. A *clash* means that a node contains a formula and its negation. A formula is satisfiable iff the model constructed by this algorithm does not contain a clash and no rule is applicable.

In [LST99], the proof is given that this algorithm terminates and can be implemented in PSpace using a reset-restart mechanism to avoid having to keep the whole model tree in memory at the same time.

For example, a tree representing a model for the formula

$$\begin{aligned}&\exists v(R(v, x) \wedge D(v)) \wedge \\ &\exists(y, z)(P(x, y, z) \wedge C(y) \wedge \exists(v, w)(P(z, v, w) \wedge C(v) \wedge D(w)))\end{aligned}$$

is displayed in Figure 1: the free variable x is replaced with the constant c, which becomes a member of the root node. Both sub-formulae of the conjunction are added to the node by the first rule. Then, the $\exists$-formulae are satisfied by the third rule, replacing v with h and y, z with d, e. After creating the next node, no more rules are applicable.

Implementation To implement this theoretical algorithm, several problems have to be dealt with:

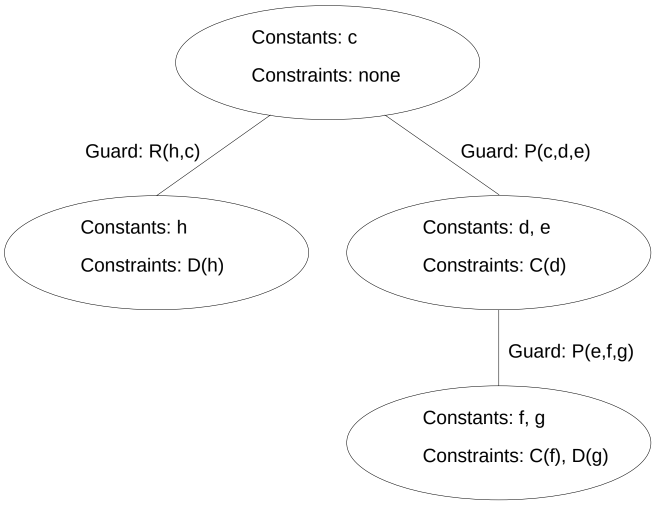

Figure 1: Example model tree for $\mathcal{GF}1^-$.

Nondeterminism To test the satisfiability of a $\vee$-formula, both alternatives have to be checked. But since $\mathcal{GF}1^-$ allows for the n-ary complement of inverse roles, and therefore a formula in a node can modify this node's predecessor, the whole model is copied before making a nondeterministic decision. Then, one alternative is added to the copy and the other one is added to the original model (*Branching*). Because branching is expensive regarding both time and space, it has to be delayed if possible, and it is important to make a good decision about which alternative to test first.

Variables Since the $\mathcal{GF}1^-$ syntax contains variables, it is possible for one single formula to appear in syntactically different forms, e.g. $\forall x_1(G(c, x_1) \rightarrow \varphi(x_1))$ and $\forall x_5(G(c, x_5) \rightarrow \varphi(x_5))$. To be efficient, it is important to detect the (semantic) equality between these formulae and a contradiction like $(\forall x_1(G(c, x_1) \rightarrow \varphi(x_1))) \wedge \neg(\forall x_5(G(c, x_5) \rightarrow \varphi(x_5)))$.

Space vs. time efficiency A reset-restart algorithm consumes time to gain space efficiency. Since for an implementation time efficiency is the most important factor, the algorithm will in fact keep the whole model in memory.

Negation normal form NNF is not helpful for $\mathcal{GF}1^-$ since it translates the obviously contradictory formula $(\varphi \vee \chi) \wedge \neg(\varphi \vee \chi)$ to $(\varphi \vee \chi) \wedge (\neg\varphi \wedge \neg\chi)$, preventing the unsatisfiability from being detected immediately. The effect on $\forall$- and $\exists$-formulae is similar.

3 Syntactic Preprocessing

To prevent the algorithm from trying to construct models for "obviously" contradictory formulae, several techniques were implemented to preprocess the formula. Most of these, in particular Early Clash Detection, Lazy Unfolding, Normalization and Encoding, are described in [Hor97], implemented in FaCT, and were adapted to $\mathcal{GF}1^-$.

Encoding, Lazy Unfolding and Early Clash Detection Each formula is assigned a natural number (*Encoding*), and as long as it a sub-formula of a surrounding formula, it is only represented by this number (*Lazy Unfolding*). A positive formula receives an even number, and its negation receives the succeeding odd number. Thus, $\varphi \wedge \neg\varphi$ is encoded as $2 \wedge 3$, which makes it possible to detect the contradiction immediately, even if φ itself is complex (*Early Clash Detection*, ECD). To encode and decode efficiently, formulae are mapped to numbers using a hash table, and numbers are mapped to formulae using an array.

Normal form For reasons explained in section 2, instead of NNF a normal form is used which allows negation of complex formulae, but only needs the junctors $\neg$ and $\wedge$ and the quantifier $\forall$. This way, the formula $(\varphi \wedge \chi) \wedge (\neg\varphi \vee \neg\chi)$ is translated to $(\varphi \wedge \chi) \wedge \neg(\varphi \wedge \chi)$, which ECD will identify as a contradiction.

Hierarchical $\wedge$-formulae, (e.g. $(\varphi \wedge \chi) \wedge (\neg\psi \wedge (\neg\varphi \wedge \omega)))$ are transformed to n-ary one-level formulae $(\bigwedge(\varphi, \chi, \neg\psi, \neg\varphi, \omega))$. The sub-formulae are represented by their code numbers $(\bigwedge(2, 4, 7, 3, 8))$, which are sorted numerically $(\bigwedge(2, 3, 4, 7, 8))$. Thus, every sub-formula only has to be compared to the subsequent one to detect a contradiction (between 2 and 3, but not between 7 and 8). Negated $\wedge$-formulae, which stand for $\vee$-formulae, are treated the same way to detect a tautology.

Syntactic Simplification Formulae containing redundant information are simplified according to the rules

$$\begin{array}{rclcrcl} \varphi \wedge \varphi & = & \varphi & \quad & \varphi \wedge \top & = & \varphi \\ \varphi \wedge \bot & = & \bot & \quad & \forall \overline{x}(G(\overline{x}, \overline{y}) \rightarrow \top) & = & \top \end{array}$$

and the dual rules for $\exists$ and $\vee$.

Variable normalization To avoid redundancy caused by separately checking formulae which differ only in their variable names (see section 2), variables are normalized to start with the smallest possible number. But we need several classes of nodes containing only variables reserved for their class to prevent ambiguity between variables quantified within sub-formulae. As every constraint contains variables from at most two nodes (each constraint is *uni-* or *bi-node* [LST99]), two sets are sufficient: for the root node, odd variable indices are used, and every node uses even numbers if its predecessor uses odd ones, and vice versa.

Thus, $\forall x_4(P(x_4, x_5) \to \exists x_6(P(x_4, x_6) \wedge \exists x_8(P(x_6, x_8) \wedge S(x_8))))$ is translated into $\forall x_2(P(x_2, x_1) \to \exists x_1(P(x_2, x_1) \wedge \exists x_2(P(x_1, x_2) \wedge S(x_2))))$.

4 Optimization

While constructing a model, especially when branching, several heuristics can be used to improve the performance. Again, these were taken from FaCT [Hor97].

Dependency Directed Backtracking Dependency Directed Backtracking, or *Backjumping* [Hor97] modifies the way backtracking is performed after a clash. Instead of backtracking to the most recent branching point, one goes back to the last branching point *one of the clashing formulae depends on.*

To do so, every branching point is assigned a number. The formulae added to the model are labeled with that number, and all formulae added by deterministic expansion inherit the labels of their predecessors. This makes it possible to determine which nondeterministic decision caused the clash.

For example, if the node n contains the formulae

$$\exists x_1(G(c, x_1) \wedge \varphi(x_1)) \tag{1}$$

$$(C(c) \vee D(c)) \tag{2}$$

$$\forall x_1(G(c, x_1) \to (\neg\varphi(x_1) \wedge \psi(x_1))) \tag{3}$$

(where c is a new constant of n), and the formulae are satisfied in the given order, 1 and 3 will clash and backtracking to 2 will inevitably result in another clash. With Backjumping, the decision for 2 receives the number 1 (because it is the first branching point), and $C(c)$ is labeled with $\{1\}$. The labels of the clashing formulae are $\varnothing$, and therefore the algorithm does not add $D(c)$ to n, but it backtracks immediately to the next upper (in this case: top) level and fails.

To maximize the efficiency of Backjumping, whenever deciding which formula to expand next, the one with the earliest Backjumping identifier is chosen.

Semantic Branching When trying to satisfy the formula $\varphi \vee \chi$, the model constructed so far is copied, and φ is added to the first copy. If this results in a clash, instead of χ, $\neg\varphi$ is added to the second copy (which later also leads to adding χ). This can improve the performance because information already gathered is not lost: the information that φ is unsatisfiable within the model is added to the other copy.

However, in a naive implementation Semantic Branching can degrade performance, e.g. when φ is $\forall x(G(x,y) \to \chi(x))$: $\neg\varphi$ translates into $\exists x(G(x,y) \wedge \neg\chi(x))$. This leads to the construction of a new node and possibly more if χ is complex. Therefore, the formula on which to branch is selected by the branching heuristics *MOMS* or *Maximize-Jump* described below.

Local Simplification and Boolean Constraint Propagation Branching on a formula whose negation is already member of a node leads to an immediate clash and wasted time for copying the model. Therefore, only *open* formulae, i.e. formulae which so far are neither true nor false within the corresponding node, are selected for branching.

In particular, if all alternatives but one would cause a clash, the remaining one is deterministically added to the model (*Boolean Constraint Propagation*, BCP). For example, if a node contains the predicates $C(c)$ and $\neg D(c)$, no branching is performed for the formula $\bigvee(\neg C(c), D(c), E(c))$, but simply $E(c)$ is added. Before branching, all disjunctions are checked for BCP applicability to delay branching as much as possible.

Branching heuristics For finding the next formula to branch on, several heuristics were used. The first one is called *MOMS*, which stands for "Maximum number of Occurrences in disjunctions of Minimum Size" [Fre95]. It works by counting the positive and negated appearances of all sub-formulae appearing in minimum-size disjunctions. It then chooses the formula φ with the largest count and adds φ and $\neg\varphi$ to the copies of the model. If the count of φ was larger than that of $\neg\varphi$, it adds $\neg\varphi$ to the first copy, otherwise it adds φ. This reduces the size of the remaining disjunctions and therefore maximizes the effect of Local Simplification and BCP.

But since MOMS checks the more constrained model first, the first alternative is more likely to fail, and therefore it may lead to additional overhead for copying a model, detecting a clash and afterwards continuing with the less constrained alternative. To test this hypothesis, we also ran the benchmark with an "inverted" MOMS algorithm which adds φ and $\neg\varphi$ in the opposite order.

Horrocks et al. observed [HST00] that MOMS does not work well together with Backjumping because the most constrained formula is not necessarily the one leading to the furthest backjump. Therefore, we also tried another heuristic: *Maximize-Jump* chooses the first alternative of the formula for which the maximum Backjumping identifier is minimal to improve the efficiency of Backjumping.

5 Empirical Results

To test the efficiency of our implementation, we used the *Logics Workbench* benchmark formulae [BHS00], which were also applied on the *TABLEAUX '98* [dS98] conference. This benchmark suite consists of nine sets of formulae named *k_branch*, *k_d4*, etc. in provable (*_p*) and not provable (*_n*) variants and different complexity levels (1–21). The efficiency of a solver is indicated by the most complex formula it can solve in less than 100 seconds.

Since $\mathcal{GF}1^-$ is not a prover, we tested if the negation of the given formula was satisfiable.

Comparison of heuristics To get a clearer overview on the efficiency of the different optimizations, we do not give the results for every single formula set, but the sum of all formulae that could be solved. The results for different combinations of heuristics are shown in table 1. "Y" in a column means that the corresponding heuristic was enabled. The numbers in brackets stand for "inverted" MOMS as described in section 4.

Syntactic Simpl.	Backjumping	Sem. Br. and BCP	MOMS	Σ
				47
Y				97
Y		Y		137
Y		Y	Y	(150) 129
Y	Y			132
Y	Y	Y		178
Y	Y	Y	Y	(183) 143

Table 1: Performance with different heuristics

Not all combinations are shown because MOMS requires Semantic Branching (it adds φ and $\neg\varphi$ to the models), and Syntactic Simplification takes too little time itself to expect a speedup if it is disabled.

The results show that Backjumping and Semantic Branching are very efficient by themselves, and that they work well together. On the opposite, MOMS slows down the solver independent of the presence of backjumping. With inverted MOMS, this effect is cancelled out, which shows that for $\mathcal{GF}1^-$, the bad performance is caused by the copying overhead rather than the interaction with backjumping. The most efficient branching heuristic also depends on the particular formula.

Comparison with other solvers Since our $\mathcal{GF}1^-$ implementation uses many of the FaCT heuristics, we compared it with FaCT and two other optimized solvers, KSAT and DLP. The results are shown in table 2. However, these results are preliminary because we did not re-run the benchmark for the other solvers, but we used results presented in [HPS98], which were produced on a different platform. We did so because our intention was not to "beat" the other solvers, but to find out if our $\mathcal{GF}1^-$-implementation scales well, i.e. if it offers a comparable performance on formulae which are written in a logic less expressive than $\mathcal{GF}1^-$.

As the formulae themselves are not tailored for $\mathcal{GF}1^-$, we have some overhead: one of the time-consuming factors is the permission of inverse roles (see section 2), which leads to the necessity of copying the whole model when branching. For the logic **K**, it would be sufficient after a clash to backtrack to the nondeterministic decision that caused the clash, delete the subtree constructed and start a new one with another alternative. Because the "t4p" formulae contain many disjunctions, we think that this overhead is the reason why the performance for this formula is so poor. But this is the only one of the benchmark formulae for which $\mathcal{GF}1^-$ is significantly slower than the other solvers.

$\mathcal{GF}1^-$ branching performance might be improved if the model was not copied, but the changes resulting from a nondeterministic decision were marked in some way such that they could be reverted after a clash. This is subject to further study.

	$\mathcal{GF}1^-$		FaCT		DLP		KSAT	
Formula	p	n	p	n	p	n	p	n
k_branch_	10	6	6	4	19	13	8	8
k_d4_	5	3	21	8	21	21	8	5
k_dum_	9	14	21	21	21	21	11	21
k_grz_	21	21	21	21	21	21	17	21
k_lin_	21	21	21	21	21	21	21	3
k_path_	5	3	7	6	21	21	4	8
k_ph_	6	9	6	7	7	9	5	5
k_poly_	17	8	21	21	21	21	13	12
k_t4p_	2	2	21	21	21	21	10	18

Table 2: Results for $\mathcal{GF}1^-$ and other solvers

Platforms: $\mathcal{GF}1^-$: Hardware: Pentium III-450, 256 MB. Software: Linux, Allegro Common Lisp 5.0, compiled. DLP: 150 MHz Ross RT626 CPU, 132 MB. Software: SML-NJ compiler, version 109.32, compiled. FaCT and KSAT: Hardware: SUN Ultra 1 (147 MHz), 32 MB. Software: Solaris, Allegro CL 4.3, compiled.

To obtain results for a more expressive logic allowing for inverse roles, we also ran $\mathcal{GF}1^-$ with the "modal QBF with inverse" set of the *TANCS 2000* [tan] benchmark formulae. It consists of groups of eight random generated formulae which are characterized by the number of clauses and variables they contain, and by their modal depth. The results for some of the easy groups (p-qbf-inv-cnfSSS-K4-Cx-Vy-Dz) are given in table 3. It shows for every group (indicated by its x, y, and z values) the number of formulae found to be satisfiable or unsatisfiable (#), the median of the calculation times (Med, in seconds), and the number of timeouts (T) and memory failures (M).

The second column shows the results of $\mathcal{SHIQ}$, which extends FaCT with inverse roles. They were taken from [Hor00].

	$\mathcal{GF}1^-$						$\mathcal{SHIQ}$					
Formula	Sat		Unsat		Fail		Sat		Unsat		Fail	
C-V-D	#	Med	#	Med	T	M	#	Med	#	Med	T	M
10-4-4	8	10.88	0	-	0	0	2	11.19	0	-	0	6
20-4-4	5	24.40	2	151.33	1	0	1	36.43	2	70.37	0	5
30-4-4	2	209.79	4	142.02	2	0	1	112.48	1	275.13	0	6
40-4-4	0	-	8	188.78	0	0	0	-	5	176.97	0	3
50-4-4	0	-	8	263.28	0	0	0	-	8	11.23	0	0

Table 3: TANCS results for $\mathcal{GF}1^-$ and $\mathcal{SHIQ}$

Platforms: $\mathcal{GF}1^-$: as above; timeout: 600 sec. $\mathcal{SHIQ}$: Hardware: Pentium III-450, 128 MB and Celeron-433, 256 MB. Software: Linux, Allegro Common Lisp, compiled; timeout: 600 sec.

In contrast to the Logics Workbench formulae, $\mathcal{GF}1^-$ is in most cases more efficient than $\mathcal{SHIQ}$. This demonstrates that the overhead caused by the possibility of inverse roles at least partially explains the inferior performance for **K** formulae.

6 Conclusion

In this paper we presented implementation details and empirical results for an algorithm deciding $\mathcal{GF}1^-$-satisfiability. It was shown that Syntactic Simplification, Semantic Branching and Backjumping significantly improve performance, while MOMS leads to additional overhead for trying the more constrained model first. Inverted MOMS leads to slightly improved performance in contrast to Maximize-Backjump, but its efficiency depends on the structure of the formula under consideration.

The overall performance in comparison with other systems is satisfactory for **K**, which contains only a small subset of $\mathcal{GF}1^-$, and good for the more expressive logic “modal QBF with inverse roles”.

References

[ANvB98] H. Andréka, I. Németi, and J. van Benthem. Modal logic and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[BHS00] P. Balsiger, A. Heuerding, and S. Schwendimann. A benchmark method for the propositional modal logics K, KT, S4. *Journal of Automated Reasoning*, 24(3):297–317, April 2000.

[dS98] H. de Swart, editor. *Automated Reasoning with Analytic Tableaux and Related Methods*, number 1397 in Lecture Notes in Artificial Intelligence, Oisterwijk, Netherlands, May 1998. Springer-Verlag.

[Fre95] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, 1995.

[Hor97] I. Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[Hor00] I. Horrocks. Benchmark analysis with FaCT. In *Proceedings of Tableaux 2000*, 2000. to appear.

[HPS98] I. Horrocks and P. F. Patel-Schneider. FaCT and DLP. In *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 27–30, 1998.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, May 2000.

[LST99] C. Lutz, U. Sattler, and S. Tobies. A suggestion for an n-ary description logic. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics*, pages 81–85. Linköping University, 1999. Proceedings online available from http://sunsite.informatik.rwth-aachen.de/publications/ceur-ws/vol-22/.

[Spa93] Edith Spaan. The complexity of propositional tense logics. In Maarten de Rijke, editor, *Diamonds and Defaults*, pages 287–307. Kluwer Academic Publishers, Dordrecht, 1993.

[tan] TANCS (tableaux non classical systems comparison) for the TABLEAUX 2000 conference
URL: http://www.dis.uniroma1.it/∼tancs/.