Implementing an $\mathcal{ALCRP}(\mathcal{D})$ ABox Reasoner – Progress Report –

Volker Haarslev and Ralf Möller and Anni-Yasmin Turhan
University of Hamburg, Computer Science Department,
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
`http://kogs-www.informatik.uni-hamburg.de/~<name>/`

Abstract: *This paper presents a progress report on the implementation of an $\mathcal{ALCRP}(\mathcal{D})$ ABox reasoner and a knowledge representation framework. We present an $\mathcal{ALC}$ ABox reasoner which has been constructed for providing a basis for an optimized $\mathcal{ALCRP}(\mathcal{D})$ implementation. We compare the implementation with the concept consistency reasoner* FaCT *which sets the standard in current DL implementations.*

Keywords: Implementation of description logic reasoners, ABox consistency checking.

1 Introduction

Extending the work on $\mathcal{ALC}(\mathcal{D})$ [1], we have developed a new description logic called $\mathcal{ALCRP}(\mathcal{D})$ [4, 8, 9] in order to provide a foundation for spatioterminological reasoning with description logics. The goal was to develop a description logic that provides modeling constructs that can be used to represent topological relations as roles. In a specific domain model, $\mathcal{ALCRP}(\mathcal{D})$ can be used to define roles (e.g. for representing topological relations) based on properties between concrete objects which, in turn, are associated to individuals via specific features. Thus, $\mathcal{ALCRP}(\mathcal{D})$ provides role terms that refer to predicates over a concrete domain ($\mathcal{RP}$ stands for **r**oles defined as **p**redicates). With these constructs $\mathcal{ALCRP}(\mathcal{D})$ extends the expressive power of $\mathcal{ALC}(\mathcal{D})$ (for a comparison, see [8]). However, in order to ensure termination of the satisfiability algorithm, we impose restrictions on the syntactic form of the set of terminological axioms [4]. Although modeling is harder, the restrictions on terminologies ensure decidability of the language.

This paper presents a progress report on the implementation of an $\mathcal{ALCRP}(\mathcal{D})$ ABox reasoner. As a first step, we describe an $\mathcal{ALC}$ ABox reasoner which has been constructed for providing a basis for an optimized $\mathcal{ALCRP}(\mathcal{D})$ implementation. Since $\mathcal{ALC}$ is a proper sublanguage of $\mathcal{ALCRP}(\mathcal{D})$, a fast reasoner for $\mathcal{ALC}$ is required as a sound basis for an $\mathcal{ALCRP}(\mathcal{D})$ implementation. Our work is inspired by Ian Horrocks' work on a TBox reasoner (called FaCT, see [7]). Although, in principle, consistency checking for $\mathcal{ALC}$ ABoxes can be reduced to TBox consistency checking [6], for practical purposes we decided to develop a true ABox consistency checker in the first place. Note that the tableau calculus for $\mathcal{ALCRP}(\mathcal{D})$ is already given for ABoxes (see [8]).

We compare our current ABox implementation (called HAM-ALC) with the concept consistency reasoner FaCT which sets the standard in current DL implementations.[1] In addition, the paper briefly presents two parts of a knowledge representation framework which have been developed based on HAM-ALC: (i) an implementation of the algorithms for embedding defaults into terminological formalisms presented in [2] and (ii) a Web interface for defining TBoxes (and ABoxes) with an infix syntax (ASCII counterpart of the "German Syntax" for LaTeX, see below).

2 The Implementation

The architecture of HAM-ALC explicitly represents the rules for a standard tableau calculus as procedures which operate on lists of concept constraints. In this section we give a sketch of the implementation details. Due to lack of space, we assume that the reader is familiar with $\mathcal{ALC}$ tableau calculi.

2.1 Preprocessing

The preprocessing phase transforms concept expressions into negation normal form, removes duplicates, performs obvious simplifications, detects clashes, flattens nested and/or expressions, normalizes the order of disjuncts and conjuncts, and ensures that all concepts, which are (structurally) `equal`, are also `eq` (pointer equality). For each concept, its negated counterpart is precomputed and accessible via a hashtable, so the computation of negations of concepts (required for clash detection, see below) is fast.

2.2 Data Structures

ABox constraints are implemented as structures (records) that hold an individual (a number), a non-negated concept, a flag `negated-p` that indicates whether the non-negated concept is to be interpreted as negated, and a set of dependency constraints (required for dependency-directed backtracking, see below). In order to ensure extensibility to $\mathcal{ALCRP}(\mathcal{D})$ we decided not to use special encoding tricks for representing concepts. Thus, concepts are represented using structures as well. We internally convert the concepts of or- and all-constraints into their equivalent negated form (e.g. $(C_1 \sqcup C_2) \rightarrow \neg(\neg C_1 \sqcap \neg C_2)$ and $(\forall\ \mathsf{R}\ \mathsf{C}) \rightarrow \neg(\exists\ \mathsf{R}\ \neg\mathsf{C})$) but represent the negation sign of the concept with the `negated-p` flag.

In contrast to FaCT and traditional SAT implementations, where a satisfiable truth assignment is represented as a vector over all known literals or constraints (as a part of the so-called *constraint system*), HAM-ALC

[1] The fact that both HAM-ALC and FaCT are implemented in Common Lisp greatly facilitated the testing and comparison.

uses a different scheme. We collect non-contradictory constraints in various lists and represent the truth value of a constraint by the negated value of its `negated-p` flag. As a consequence of this decision, there is no need to save and restore constraint systems during backtracking due to the pointer representation of lists in Lisp.

2.3 Clash Detection

As usual, the first step of the tableau algorithm is to detect whether adding a new element (i.e. assigning a truth value) to the set of expanded constraints (that already have an assigned truth value) will result in a clash. Since concepts can be compared with `eq` and constraints contain only non-negated concepts as well as the `negated-p` flag, this operation does not require to traverse concept structures. However, clash detection is currently more expensive —compared with traditional SAT implementations— because the new constraint has to be checked against the set of all expanded constraints.

2.4 Tableau Control Strategy

In contrast to the theoretical formulation of tableau calculi, the order of processing and applying tableau expansion rules to constraints makes a big difference in the efficiency of a tableau algorithm. HAM-ALC currently supports two principal control strategies. With the "true $\mathcal{ALC}$" setting HAM-ALC processes the constraints in the following order: (1) deterministic constraints, (2) or-constraints, (3) some-constraints. Deterministic constraints are those constraints whose concept term is either atomic, an and-concept, or an or-concept with exactly one open disjunct (i.e. it has an unknown truth value).

This processing order and the fact that the tree model property holds for $\mathcal{ALC}$ allows a well-known optimization for $\mathcal{ALC}$, where consistency of concept constraints for individuals generated by some-constraints can be treated as isolated subproblems if value restrictions are carefully treated. This implies the expansion of some-constraints with a separated *empty* constraint system. The "true $\mathcal{ALC}$" setting was also used for benchmarking HAM-ALC for the DL comparison (see [5]).

The second setting is designed for $\mathcal{ALCRP}(\mathcal{D})$ and always works with *one* global constraint system because, for $\mathcal{ALCRP}(\mathcal{D})$, the tree-model property does not hold [4]. The ABox consistency checker has to deal with graph structures at least in a finite part of the ABox. Thus, we also have to explicitly represent relation constraints.

As already noted, the constraint system used in HAM-ALC is represented as a collection of lists. We use five different lists: (1) unexpanded deterministic constraints, (2) unexpanded or-constraints, (3) unexpanded some-constraints, (4) expanded concept constraints, (5) expanded relation constraints. The elements from the "unexpanded lists" are always processed in accordance with the priority scheme explained above.

2.5 Optimization Techniques

Or-constraints are a major source of complexity in tableau expansion. Due to the experiences of the FaCT system, two major optimization strategies (adapted from the SAT domain) are absolutely necessary to achieve acceptable performance even for small problems (see [3], [7]). The first technique is called *semantic branching*, the second one is *dependency-directed backtracking*.

Semantic Branching

In contrast to syntactic branching, where redundant search spaces may be repeatedly explored, semantic branching uses a *splitting rule* which replaces the original problem by two smaller subproblems (see also [3]). Semantic branching is usually supported by various techniques intending to speed-up the search.

A *lookahead algorithm* or *constraint propagator* tries to reduce the order of magnitude of the open search space. Thus, after every expansion step HAM-ALC propagates the truth value of the newly added constraint into all open disjuncts of all unexpanded or-constraints. As a result of this step, or-constraints might be become satisfied (i.e. one disjunct is satisfied), deterministic (i.e. exactly one disjunct remains open), or might even clash (all disjuncts are unsatisfied).

Various *heuristics* are used to select the next unexpanded or-constraint and disjunct. HAM-ALC employs a dynamic selection scheme. The oldest-first strategy is used for selecting one or-constraint with at least two open disjuncts. In accordance with FaCT we also count the negated and unnegated occurrences for each open disjunct from the selected constraint in all other unexpanded or-constraints. These numbers are used as input for a priority function that selects the disjunct. The priority function is adopted from FaCT and achieves the following goals. It prefers disjuncts that occur frequently in unexpanded binary constraints and balanced or-constraints (i.e. containing a similar number of negated and unnegated occurrences of the same disjunct) but discriminates between unbalanced or-constraints. In order to perform the counting very quickly, HAM-ALC precomputes for every constraint two lists cross-referencing or-constraints that contain the constraint's concept in negated or unnegated form. Once a disjunct is selected, the priority function is also used to determine whether the constraint is tried in negated or unnegated form at first. In case of a failure, the other alternative is explored.

Dependency-directed Backtracking

Naive backtracking algorithms often explore regions of the search space rediscovering the same contradictions repeatedly. Dependency-directed backtracking records the dependencies of expanded constraints and in case of a clash backtracks to (or-)constraints that are responsible for at least one of the clash-causing constraints in the subtree (see [3]). Experiments with FaCT, HAM-

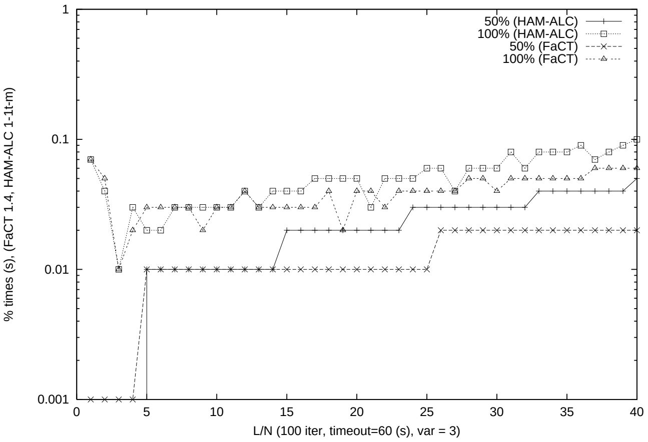

Figure 1: Comparison of runtime for $N = 3$.

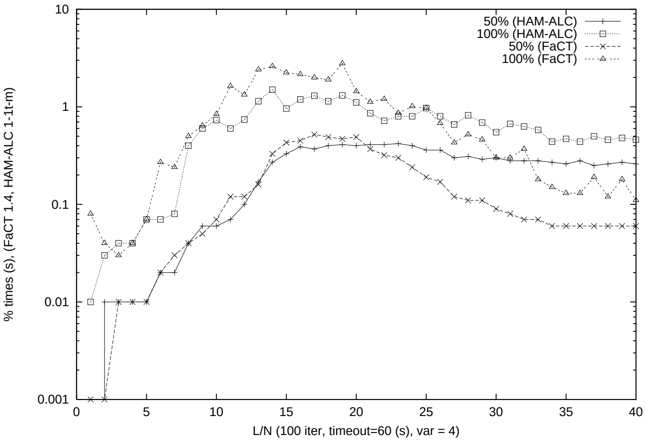

Figure 2: Comparison of runtime for $N = 4$.

ALC, and other DL systems demonstrated that disabling dependency-directed backtracking ("backjumping") does not make much sense and, therefore, the dependency management system is wired into the HAM-ALC architecture and its data structures. When a constraint is expanded, its dependencies are recorded, i.e. its precondition constraints are pushed onto the list of dependencies. In our implementation, every or-constraint on the dependencies of a precondition constraint is also pushed onto the list of dependencies of the resulting constraint (dependency propagation).

When a clash occurs, the dependencies of the clash culprits (i.e. possibly the or-constraints that generated these constraints) are stored on a list of catching clash dependencies and backtracking is started. The idea is due to FaCT [7]. Whenever a semantic branching point is encountered during backtracking, HAM-ALC checks whether this or-constraint is responsible for a clash culprit. If the or-constraint is not found in the list of catching clash dependencies, we can safely bypass this branching point. In case the or-constraint is found either the remaining semantic alternative is tried or this disjunct is considered as unsatisfiable in the current subtree. The backtracking continues but removes the current or-constraint from the list of clash dependencies and adds the saved clash dependencies of the previously unsatisfiable alternative.

3 Evaluation of HAM-ALC

The evaluation examples presented in this paper are guided by the examples from [7]. The FaCT system is used with standard settings as provided by Horrocks. Thus, concept normalization, backjumping and semantic branching are enabled.

We conducted three experiments[2] using concept terms randomly generated by the "Hustadt and Schmidt" generator (for details see [7]). The generator is controlled by the following set parameters (fixed values are printed in parentheses):

- Number of different primitive concepts: N
- Number of different roles: $M(= 1)$
- Size of K-disjunctive expressions: $K(= 3)$
- Maximum nesting of value restrictions: $D(= 1)$
- Probability of a disjunct being a literal: $P(= 0)$
- Number of K-disjunctive expressions in the top-level conjunction: L

Each experiment used a fixed value for N ($N = 3, 4, 5$) and increased the value of L from N to $N * 40$ with N as increment. The results of the first experiment ($N = 3$, 100 tests for each value of L) are displayed in Figure 1. The graphics present the percentile runtime[3] of FaCT and HAM-ALC, respectively. FaCT behaves slightly better than HAM-ALC but the difference between them remains almost constant.

The second experiment[4] ($N = 4$, 100 tests for each value of L) demonstrates a different behavior (see Figure 2). HAM-ALC outperforms FaCT almost from the beginning but FaCT still succeeds for values of $L/N > 25$. This is probably due to the clash-detection technique of HAM-ALC that currently requires searches in lists of increasing sizes with an increasing L. We also measured the storage requirements (percentile memory) of both systems (see Figure 3). In the transition phase FaCT requires almost one order of magnitude more memory than HAM-ALC.

The last experiment ($N = 5$, 10 tests for each value of L, 1000 seconds as timeout limit) demonstrates the beginning combinatorial explosion for both FaCT and HAM-ALC (see Figure 4). However, HAM-ALC solves

[2]The tests were performed on an Ultra Sparc 2 with Allegro Common Lisp 4.3.1 and both systems compiled with full optimization.

[3]The 100th percentile is the time taken to solve the hardest problem of the test set. The 50th percentile represents the time taken by the problem in the test set such that 50 percent of the other problems took a shorter runtime [7].

[4]Also called PS12 in [7].

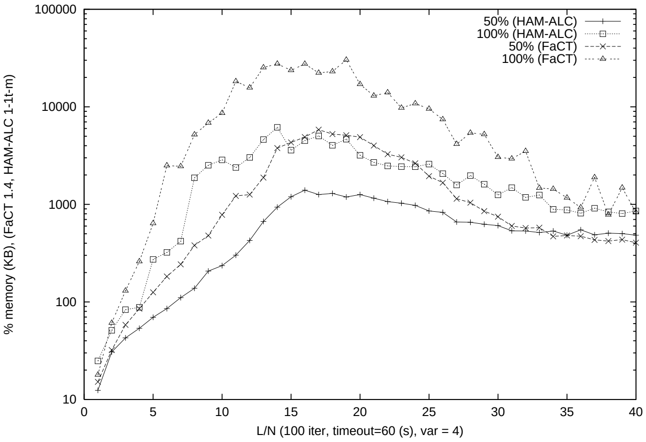

Figure 3: Comparison of space for $N = 4$.

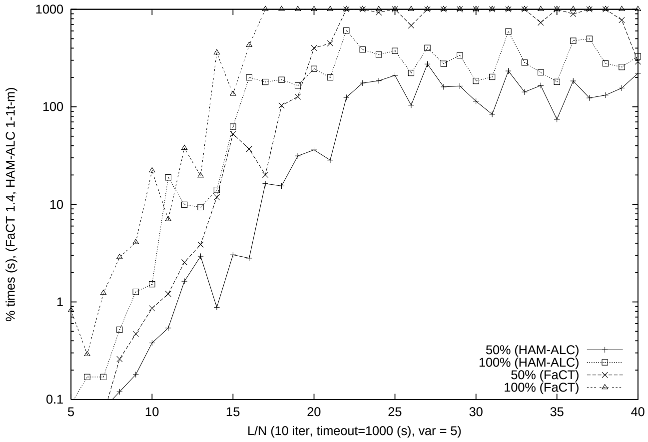

Figure 4: Comparison of runtime for $N = 5$.

even the hardest problems within 600 seconds.

To the best of our knowledge, not very much work has been published on evaluating ABox consistency checkers. So, this is currently an open problem.

4 Towards a Modeling Framework

An ABox consistency checker provides the basic algorithm for other reasoning services. For instance, default rules can be integrated into a knowledge representation framework.

4.1 Defaults

We implemented a Reiter-style default rule system according to the theory and algorithms for computing extensions presented in [2]. Default rules are applied to ABox individuals. With this system, for instance, the famous Nixon example can be declared as follows. Preconditions and conclusions are $\mathcal{ALC}$ concepts, justifications are lists of $\mathcal{ALC}$ concepts.

```
(define-default-rule nixon-as-a-republican
  :precondition nixon :justifications () :conclusion republican)

(define-default-rule nixon-as-a-quaker
  :precondition nixon :justifications () :conclusion quaker)
```

Two contradictory defaults are added.

```
(define-default-rule quakers-as-doves
  :precondition quaker :justifications ((not hawk))
  :conclusion dove)

(define-default-rule republicans-as-hawks
  :precondition republican :justifications ((not dove))
  :conclusion hawk)
```

The example is completed with the following defaults.

```
(define-default-rule doves-are-politically-motivated
  :precondition dove :justifications ()
  :conclusion politically-motivated)

(define-default-rule hawks-are-politically-motivated
  :precondition hawk :justifications ()
  :conclusion politically-motivated)
```

The default substrate computes the extension of an ABox and a set of default rules (a) in terms of sets of applicable "non-contradictory" closed default rules and (b) in terms of sets of ABoxes ("possible worlds") resulting from applying each set of non-contradictory default rules. For instance, if we start with an ABox `((i nixon))` and the set of defaults presented above, the system computes the following extensions:

```
Extension 1:
Defaults: nixon-as-a-quaker(i), nixon-as-a-republican(i),
          republicans-as-hawks(i), hawks-are-politically-motivated(i)
ABox: ((i quaker) (i republican) (i hawk) (i politically-motivated))

Extension 2:
Defaults: nixon-as-a-quaker(i), nixon-as-a-republican(i),
          quakers-as-doves(i), doves-are-politically-motivated(i)
ABox: ((i quaker) (i republican) (i dove) (i politically-motivated))
```

Thus, as expected, the `Nixon` individual `i` will be politically motivated for credulous as well as for skeptical reasoners.

The short example should demonstrate the available facilities which go beyond those offered by standard rule systems available in description logic implementations. The dependency tracking mechanism of the HAM-ALC ABox reasoner is an important part of the implementation.

4.2 Web Interface

Especially for teaching purposes, a simple interface is required for specifying TBoxes and ABoxes etc. Thus, we extended our HAM-ALC environment with a Web interface (see Figure 5) supporting password access and persistency of (multiple) TBoxes (and ABoxes) as well as queries. Knowledge bases can be sent back to users via email. We also developed a parser for an infix syntax for DLs (and PDLs, see Figure 5) supporting error messages (with line numbers).

5 Summary and Outlook

With the current HAM-ALC implementation a good testbed for studying optimization techniques has been developed. The ABox consistency prover architecture of HAM-ALC should support the extensions required for $\mathcal{ALCRP}(\mathcal{D})$. As a summary we emphasize the following points:

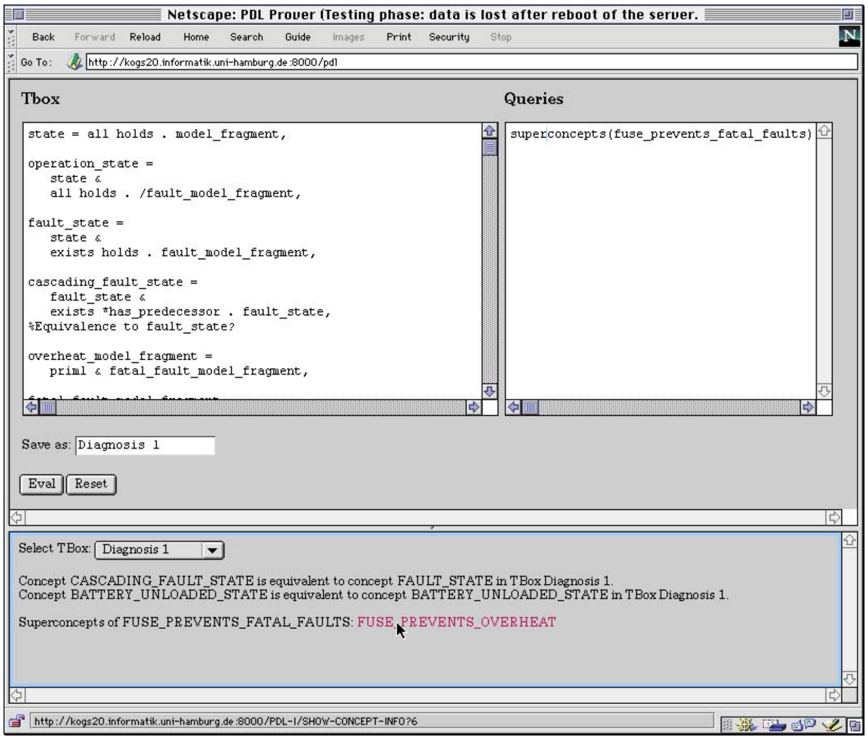

Figure 5: WWW interface supporting an infix syntax for $\mathcal{ALC}$ (the screenshot actually shows a version that is based on a PDL prover written in C).

- We implemented a consistency checker for ALC ABoxes in Common Lisp using semantic branching and dependency-directed backtracking for optimizing runtime performance.
- The default substrate (see above) will be extended by facilities for declaring preferences.
- The Web interface currently supports TBox declarations and subsumption queries but will be extended in order to deal with ABoxes and defaults.
- We are investigating implementation techniques and optimization strategies for dealing with concrete domains, i.e. for languages such as $\mathcal{ALCRP}(\mathcal{D})$ (and $\mathcal{ALC}(\mathcal{D})$, cf. also the work on the $\mathcal{KRIS}$ system).

Acknowledgments

The theoretical work on the $\mathcal{ALCRP}(\mathcal{D})$ tableau calculus has been done in collaboration with Carsten Lutz. We also acknowledge Ian Horrocks' work on the FaCT system that proved to be a very valuable system as verifier and for studying description logic prover implementations.

References

[1] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Twelfth International Conference on Artificial Intelligence, Darling Harbour, Sydney, Australia, Aug. 24-30, 1991*, pages 452–457, August 1991.

[2] F. Baader and B. Hollunder. Embedding defaults into terminological knowledge representation formalisms. In B. Nebel, Ch. Rich, and W. Swartout, editors, *Third International Conference on Principles of Knowledge Representation*, pages 306–317, October 1992.

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

[4] V. Haarslev, C. Lutz, and R. Möller. Foundations of spatioterminological reasoning with description logics. In T. Cohn, L. Schubert, and S. Shapiro, editors, *Proceedings of Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Trento, Italy, June 2-5, 1998*, June 1998. In press.

[5] V. Haarslev, R. Möller, and A.-Y. Turhan. HAM-ALC. In E. Franconi et al., editors, *Proceedings of the International Workshop on Description Logics (DL'98), June 6-8, 1998, Trento, Italy*, June 1998. Benchmark results for DL'98 comparison, in press.

[6] B. Hollunder. *Algorithmic Foundations of Terminological Knowledge Representation Systems.* PhD thesis, University of Saarbrücken, Department of Computer Science, 1994.

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

[8] C. Lutz, V. Haarslev, and R. Möller. A concept language with role-forming predicate restrictions. Technical Report FBI-HH-M-276/97, University of Hamburg, Computer Science Department, 1997.

[9] R. Möller, V. Haarslev, and C. Lutz. Spatioterminological reasoning based on geometric inferences: The $\mathcal{ALCRP}(\mathcal{D})$ approach. Technical Report FBI-HH-M-277/97, University of Hamburg, Computer Science Department, 1997.