Software Abstractions for Description Logic Systems

Michael Wessel and Ralf Möller

Hamburg University of Technology
Institute for Software, Technology, and Systems (STS)
Hamburg, Germany

Abstract. We explain the basics of description logics and tableau provers for reasoning with them. The implementation of tableau provers is a complicated matter and demanding from a software engineering point of view. We present and discuss their implementation in COMMON LISP and motivate and introduce some novel software abstractions for description logic system construction, which are embodied in the MIDELORA toolkit / framework.

1 Introduction & Motivation

In this paper we present some novel problem-specific software abstractions (in COMMON LISP) for the construction of description logic (DL) systems [1, Chapter 7 – 9]. These abstractions are embodied in the MIDELORA toolkit for DL system construction. The MIDELORA framework was tailored to alleviate a number of problems designers of description logic systems face. As such, one could argue that the target audience for the framework is rather small. We believe, however, that the MIDELORA software abstractions can be valuable for the broader audience of developers of semantic information processing technology in COMMON LISP. State-of-the-art DL systems (such as RACERPRO) are very complex software artefacts, and it became apparent that software engineering aspects deserve special attention. Let us start by introducing some background terminology; we assume basic knowledge in first-order predicate logic (FOPL) and general COMMON LISP familiarity only.

Description Logics & Semantic Web Description logics [1, Chapter 2] are a family of logic-based decidable knowledge representation languages which provide the formal foundation for current W3C standards for the Semantic Web, such as OWL DL. DLs can also be characterized as *class or concept-based representation languages*; i.e., the notion of a *concept or class* is a central one. A concept is meant to *denote* a set of individuals in some universe of discourse. There is also the notion of binary relationships, which are called *roles*. The set of individuals denoted by a concept is also called the *extension* of the concept, in contrast to the concept description, which is sometimes called the *intension* of the concept. Concepts are *described* in a formal language. For example, the extension of the concept described by "female and person" is given by the intersection of the set of female objects (the extension of the concept "female"), and of the set of persons (the extension of the concept "person"). As such, the concept description "female and person" can be understood as the *definition of the concept woman*. A *concept definition* thus assigns a *concept name* to a *concept description.* Obviously, a concept name is a concept description as well. A concept definition can be complete, or only "primitive". In the former case, the definition provides *necessary and sufficient conditions* for class membership, e.g. as for "woman". In this case, the concept is called a *defined concept*. Logically, this is a bi-implication or "if and only if". This corresponds to the following simple *axiom* in FOPL: $\forall x.(female(x) \wedge person(x) \leftrightarrow woman(x))$. Knowing that "Betty" is "female" and "person", we conclude that she is a "woman": $\{person(betty), female(betty)\} \models \{woman(betty)\}$, and vice versa. In the case of a *primitive concept definition*, only *necessary conditions* are provided by the description. For example, "mother" implies "parent", but not vice versa; in this case, "mother" is a *primitive concept:* $\forall x.(mother(x) \rightarrow parent(x))$. It is obvious that "mother" could become a defined concept as well if a more complete description for "mother" was provided, e.g. "woman with a child": $\forall x.(woman(x) \wedge \exists y.(has\_child(x, y) \wedge person(y)) \leftrightarrow mother(x))$.

The nowadays practically relevant DLs can be seen as subsets of FOPL in a variable-free syntax. The given FOPL axioms take the following form in DL syntax:

$$woman \doteq person \sqcap female, mother \dot{\sqsubseteq} parent, \text{ or } mother \doteq woman \sqcap \exists has\_child.person.$$

There is also a universal quantification which allows, for example, to define the concept

$$mother\_of\_only\_male\_children \doteq woman \sqcap \exists has\_child.person \sqcap \forall has\_child.male,$$

or in FOPL:

$$\forall x.(mother\_of\_only\_male\_children(x) \leftrightarrow woman(x) \wedge \exists y.has\_child(x,y) \wedge \forall y.(has\_child(x,y) \rightarrow male(y))).$$

As mentioned above, there is no single DL, but a whole *family* of DLs. DLs are distinguished from one another by their expressive power. The expressive power of a DL is influenced by the set of concept constructors it admits, and also by the kinds of axioms offered. For example, some DLs offer disjunction, whereas other do not; some offer transitive roles, etc. A concrete example for a DL ($\mathcal{ALC}$) is given below. The set of concept descriptions a $\mathcal{DL}$ admits is just a formal language, here denoted as $\mathcal{L}_{\mathcal{DL}}$. A "bigger" DL $\mathcal{DL}'$ simply admits more concept descriptions: $\mathcal{L}_{\mathcal{DL}} \subseteq \mathcal{L}_{\mathcal{DL}'}$.

Since the advent of the Semantic Web, the importance of DLs and DL systems is ever increasing. The idea of the Semantic Web is to assign machine-processable and -understandable *explicit semantics* to web resources by publishing annotations about the URIs of these web resources. These annotations are written in an XML-based description logic: OWL. OWL DL class (descriptions) can be translated into description logic concepts; this also holds for the axioms. Due to their model theoretic semantics, these annotations imply ("$\models$") a whole logical theory, representing *implicit information* about the web resource(s). This implicit information must be taken into account, for example, for query answering on the Semantic Web. Logical inference or (deductive) reasoning is often coined as the process that makes "implicit information explicit". This is the role of a reasoner, which implements an *inference algorithm;* an API is offered to access the reasoning (or inference) services of that inference mechanism. The most basic reasoning service is the *consistency* or *satisfiability checking service* which checks whether a given concept description is non-contradictory. Thus, a decision problem must be solved by the reasoner. For example, given the axioms discussed above, the concept description $woman \sqcap \neg female$ is obviously unsatisfiable or contradictory.

Tableau Calculi, ABoxes, Query Answering, Optimizations A popular and very successful class of inference algorithms for deciding the (concept) satisfiability problem for DLs (and thus, also for OWL DL) is given be the class of *tableau calculi* [1, Chapter 9]. State-of-the-art DL systems and OWL reasoning systems (such as RACERPRO, PELLET and FACT++) implement tableau calculi for a number of good reasons – they are modular and flexible (they can cover a large set of different DLs from the DL family), they are well-understood, they are intuitive, they allow for optimizations, and so on. Tableau calculi can not only decide the concept satisfiability problem for DLs, but also decide the consistency (satisfiability) problem for *extensional databases:* a DL system not only manages and reasons about concept descriptions and their definitions, but also manages extensional data. An extensional database is also called an *ABox (assertional box).* For example, the ABox $\{woman(betty), (\neg female)(betty)\}$ is contradictory given the axioms discussed above. A DL system can thus be seen as a "deductive database" which exploits reasoning to ensure data quality, and, more importantly, to *derive its query answers.* Given the extensional data $\{person(betty), female(betty)\}$ in combination with the background knowledge regarding $woman$, it is clear that $woman(betty)$ is a logical consequence. Note that this fact is *not explicit* in the extensional data. Thus, $betty$ is an answer to the *instance retrieval query* for instances of concept $woman$. The instance retrieval problem is an important *extensional* standard inference problem. Unlike in standard (relational) database systems, a DL system uses the *open world assumption* which means that the absence of information is not interpreted as negative information. The extensional "data" in a DL system is thus not considered to be "complete". In contrast, the so-called *closed world assumption* is used in standard (relational) databases. Nowadays, scalability of query answering for DL / OWL reasoning systems is a hot research topic [3]. Considering the large volume of data available on the Web, this is not surprising. However, scalability can only be achieved if *dedicated optimizations* [1, Chapter 9] are incorporated into a tableau reasoner. Unfortunately, these optimizations complicate tableau reasoners quite a bit, since certain optimization "tricks" can only be applied in certain situations for certain sublanguages which must be detected automatically, etc.

Motivation for MIDELORA MIDELORA is motivated as follows (see also [5, 4]):

1. Flexibility: as mentioned, DLs provide a whole family of logics, and thus a great deal of flexibility. In principle, it is possible to chose the right DL for the problem at hand, provided the logic is decidable. Different family members vary w.r.t. their expressivity and computational characteristics; it is well-known that expressivity and computational complexity of a logic are co-variant properties. A toolkit for the construction of DL systems should thus obviously not fix the constructible DLs. Instead, it should provide

the required components and building blocks (e.g., DL concept constructors, different kinds of axioms etc.), from which specific DL systems can be constructed, preferably through adaptation, configuration and composition - reuse is important here.

In principle, a DL system implementing a very expressive $\mathcal{DL}$ can also supply the reasoning services for a less expressive DL $\mathcal{DL}'$ if $\mathcal{L}_{\mathcal{DL}'} \subseteq \mathcal{L}_{\mathcal{DL}}$ holds. However, reasoning in $\mathcal{DL}'$ will in many cases be much more efficient, either due to a smaller computational complexity, or due to *applicable optimization techniques* which have been developed (and which may only be valid) for $\mathcal{DL}'$. Consequently, this implies that the $\mathcal{DL}$-reasoner should recognize and handle input problems in $\mathcal{DL}'$ in an optimized way. However, having to recognize and interweave the applicable optimization techniques for sublanguages complicates the implementation of the reasoner, adding yet another level of software complexity. This strongly motivates the second point (see below).

Often, flexibility is not only required w.r.t. the family of supported DLs. Also the *ABox representation is subject to change.* For example, certain applications may require persistent ABoxes, whereas others need ABoxes in which the individuals have spatial characteristics,e.g., in order to represent polygons in a map. Slightly simplified, *an ABox can be seen as a node and edge-labeled graph.* The same statement also holds for similar semi-structured data models, such as RDF. From this perspective, the ABox $\{woman(betty), has\_child(betty, charles)\}$ is simply a graph with two nodes $betty$ and $charles$, connected via an edge. The node $betty$ is labeled with the set of concept descriptions $\{woman\}$, and the edge with the set of roles $\{has\_child\}$. Considering such a graph structure as a rather conceptual data model, different logical and physical aspects of this model might be subject to change and thus require flexibility in the software architecture.

MIDELORA introduces its own generic, graph-based data model. A *substrate* is an instance of a CLOS class `substrate`, representing a graph. Different logical and physical (e.g., representational aspects) of the data model can be addressed by introducing `substrate` (`node`, `edge`, ...) subclasses. For example, an ABox is an instance of a specialized substrate class `abox`. A `map` is a specialized substrate whose nodes are instances of spatial datatypes (e.g., polygons). We also use multiple inheritance; for example, the substrate class `map-abox` has both `abox` as well as `map` as superclasses. Consequently, its nodes are ABox individuals which also have spatial characteristics. Since the physical representation and index structures used for the different kinds of substrate graphs vary drastically, the whole access to and management of the graph structure has to go through a generic "substrate protocol" which offers iterators etc., abstracting from the details of the physical representation (data abstraction).

2. Comprehensibility, problem adequateness and conciseness (and thus maintainability) of the reasoner implementations: Mathematical tableau calculi are very concise, but intellectually demanding artefacts. The software abstractions used for their implementations should thus mirror the conceptual notions found in their mathematical definitions as close as possible. For example, there is the notion of a *tableau rule* in a tableau calculus. MIDELORA thus provides an appropriate rule definition macro `defrule`. Given a close correspondence between implementation and mathematical notions, software can be seen as an "executable specification" and its comprehensibility and thus maintainability is greatly enhanced, if software abstractions are chosen in such a way that one does not get "lost in software complexity". A high-level perspective employing problem-specific software abstractions thus has to be established such that the underlying mathematical structures of tableau calculi are clearly mirrored in the source code.

In order to prevent tableau provers to grow into monsters of complexity, MIDELORA uses a unique approach. Instead of offering one complex and highly optimized prover capable of handling one very expressive DL (with different optimizations for various sublanguages), MIDELORA provides specialized, much smaller and thus much more comprehensible dedicated provers for these sublanguages, which can be maintained relatively independently from one another. Thus, instead of providing one big and complex prover with complicated optimizations for sublanguages, MIDELORA provides many small provers. The dedicated optimization techniques are moved into specialized dedicated provers for these sublanguages. Obviously, the maintainability and comprehensibility of a large number of small provers is only enhanced if appropriate software abstractions are provided, and if *common components* in these provers can be shared and reused. This is, for example, the case for tableau rules, see below.

3. Enable reuse via inheritance, adaptability and configurability: As just illustrated, reuse by inheritance is an important software organization / structuring principle in this specific domain as well. Appropriate abstractions and reusable components such as tableau rules have to be identified. The implementation

of a highly optimized tableau rule can be very complex; thus, a rich set of standard tableau rules has to be provided. Often, specialized provers require simple adaptations of standard tableau rules only. Consequently, it should be possible to "inherit and adapt" the behavior of a tableau rule, exploiting the standard "Open-Closed Principle", or simply to adapt the rule by configuring it appropriately via parameters.

The rest of this paper is structured as follows. First we define the MIDELORA space in which the reuse and inheritance for MIDELORA software abstractions is organized. Then, the DL $\mathcal{ALC}$ and a tableau calculus for $\mathcal{ALC}$ are formally introduced. A simple COMMON LISP implementation of that calculus is presented and discussed. This MIDELORA version of an extended version of this calculus is then discussed. Then comes the conclusion.

2 The MIDELORA Design Space

In order to organize the inheritance and reuse-space, the MIDELORA space is defined. The *prover* is a central notion in MIDELORA. MIDELORA allows for the definition of specialized provers for certain tasks, working on specialized substrates. *Provers* are conceived to cover regions in the three-dimensional MIDELORA *space*:

Definition 1 (MIDELORA **Space).** *The* MIDELORA *space is the Cartesian product $\mathcal{S} \times \mathcal{L} \times \mathcal{T}$, where $\mathcal{S}$ is the set of* substrate classes*, $\mathcal{L}$ is the set of supported DLs, and $\mathcal{T}$ is a set of* prover tasks.

For example, $\mathcal{T}$ can contain the DL *standard inference problems* [1, Chapter 2.2.4]: $\mathcal{T} = \{\mathsf{abox\_consistent?}, \mathsf{concept\_instances}, \ldots\}$. Substrates and languages are modeled as CLOS classes. The elements of $\mathcal{L}$ are called *language classes* in the following. A MIDELORA *prover* is a ternary CLOS (multi-) method `prover`, with arguments $(S, DL, T) \in \mathcal{S} \times \mathcal{L} \times \mathcal{T}$.

If inheritance is exploited between the elements in the sets $\mathcal{S}$, $\mathcal{L}$, and $\mathcal{T}$, a single MIDELORA prover defined for a point (S, DL, T) can in principle cover a whole region in the MIDELORA space. But this raises the question how to order the elements in the sets $\mathcal{S}$, $\mathcal{L}$, $\mathcal{T}$ w.r.t. the subclass (inheritance) relation.

Let us consider the $\mathcal{L}$ axis. As explained, a prover for $\mathcal{DL}'$ and task T can in principle also solve T for $\mathcal{DL}$, if $\mathcal{L}_{\mathcal{DL}} \subseteq \mathcal{L}_{\mathcal{DL}'}$ holds. However, a dedicated prover for $\mathcal{DL}$ is sanctioned and reasonable if $\mathcal{DL}$ can be optimized much better, and thus its implementation becomes more specialized and tailored towards $\mathcal{DL}$. Then it becomes reasonable not to intermix its implementation with $\mathcal{DL}'$. But in principle it is not desirable of having to define one dedicated prover for each point in this space (since there would be no reuse by means of inheritance).

Basically, there are two options to organize the inheritance: a) the subclass relation among the $\mathcal{L}$ axis can be co-variant w.r.t. $\subseteq$, or b) contra-variant w.r.t. $\subseteq$. Assuming there is a "super DL" prover for $\mathcal{DL}'$ which is a very expressive DL, one prover would be sufficient for the whole family. In case a dedicated prover for $\mathcal{DL}$ is reasonable, this prover would automatically be selected for input problems in $\mathcal{DL}$. However, even if there is no such dedicated prover, $\mathcal{DL}'$ could still be used and would be selected automatically by the standard CLOS dispatch mechanism. We just have to define the language class $\mathcal{DL}$ as a subclass of the language class for $\mathcal{DL}'$ (co-variant w.r.t. $\subseteq$).

Unfortunately, this organization of the language classes has a drawback. The problem is caused by the requirement that *characteristic properties* of the DLs shall be represented as *mixin classes* in the language classes. For example, the DL $\mathcal{ALC}_{\mathcal{R}^+}$ has the property "needs blocking", but $\mathcal{ALC}$ doesn't (it is not important for this discussion to understand what this property means).[1] So, mixing in the `needs-blocking` superclass for the $\mathcal{ALC}_{\mathcal{R}^+}$ language class and making the $\mathcal{ALC}$ language class a subclass would falsely inherit the `needs-blocking` property to $\mathcal{ALC}$ as well, or non-monotonic inheritance would be needed (which is even more tricky). Inheriting *non-properties*, e.g. `doesnt-need-blocking`, is not a good idea either. Thus, we have modeled the DL classes in a contra-variant way. Unfortunately, this complicates reuse then: Suppose there is a prover for $\mathcal{ALC}$ only, but no prover for $\mathcal{ALC}_{\mathcal{R}^+}$. Now a problem in $\mathcal{ALC}_{\mathcal{R}^+}$ shall be solved. Standard CLOS dispatch would then falsely select the $\mathcal{ALC}$ prover, which would be incomplete or even fail on $\mathcal{ALC}_{\mathcal{R}^+}$ input! Consequently, a prover for $\mathcal{ALC}$ as well as for $\mathcal{ALC}_{\mathcal{R}^+}$ is *required*, and reuse is *disabled* then, since a dedicated prover for each language class would be needed. Provers would

[1] For DL experts: This is a slightly simplified example. Of course, due to general axioms in the TBox, also $\mathcal{ALC}$ eventually requires blocking. But then the language analysis component would have classified this knowledge base as $\mathcal{ALC}_{\mathcal{R}^+}$, not as $\mathcal{ALC}$.

no longer be able to cover regions in the MIDELORA space. Standard CLOS dispatch thus cannot be used w.r.t. the $\mathcal{L}$-axis. We have therefore implemented an algorithm which *downcasts* the language class instance (describing the DL of the input) along the language class hierarchy until a prover is found which is defined for this (downcasted) language class. Finally, the process reaches the maximal expressive "super DL" $\mathcal{DL}'$ at the bottom of the class hierarchy. It would be interesting to implement this custom dispatch algorithm directly within CLOS.

Regarding the $\mathcal{S}$ axis, it is obvious that standard CLOS dispatch can be used. For example, a `persistent-abox` is a subclass of `abox`. A prover must be written in such a way that it will work on specialized ABoxes, i.e., subclasses of `abox`. Consequently, all access to the physical graph representation must go through the generic interface the substrate layer provides (the substrate protocol). Dynamic binding is thus also required for iterator macros, e.g., `loop-over-nodes`. For example, in case of a persistent ABox, this iterator has to fetch and construct CLOS substrate nodes on the fly from information stored in a relational database. In other cases, the macro can be expanded in a hash table iterator, and so on. Appropriate techniques for achieving this kind of data abstraction from physical representation are well known.

Finally, regarding the $\mathcal{T}$ axis, we use simple `eql` symbol dispatch, since it is not clear how to establish reuse by inheritance between elements in $\mathcal{T}$. However, often a prover task is *reducible* to another prover task. For example, in order to check whether the individual $betty$ from the ABox $\mathcal{A}$ is an instance of $woman$ (this is the *instance checking* task, $T =$ individual_instance?, which is also performed during *instance retrieval*), the prover for the task $T =$ abox_consistent? on the ABox $\mathcal{A} \cup \{betty : \neg woman\}$ can be called. The former prover then just returns `t` if the latter one returned `nil`, and vice versa. Thus, reuse can be organized by means of delegation, but not via inheritance.

3 Tableau Calculi from the Mathematical Perspective

Let us first define syntax and semantics of the basic DL $\mathcal{ALC}$ more formally. Unfortunately, the semantics of $\mathcal{ALC}$ and the central notion of *satisfiability* has to be defined formally first, since otherwise it is left unclear what a tableau calculus is all about, i.e., the problem to be decided would be left unspecified.

The syntax of concept descriptions (concepts for short) is defined inductively as follows. Let $\mathcal{N}_\mathcal{C}$ be a set of concept names ("atomic concepts"); there are 2 special concept names, called top and bottom: $\{\top, \bot\} \subseteq \mathcal{N}_\mathcal{C}$. Let $\mathcal{N}_\mathcal{R}$ be a set of binary relation names, the set of so-called roles. Each $CN \in \mathcal{N}_\mathcal{C}$ is a concept. Moreover, if C and D are concepts, and $R \in \mathcal{N}_\mathcal{R}$ is a role, then the following expressions are concepts as well: $\neg C$ (full negation), $C \sqcap D$ (conjunction), $C \sqcup D$ (disjunction), $\exists R.C$ (existential role restriction), and $\forall R.C$ (universal role restriction). A *TBox* is a set of axioms of the form $C \dot{\sqsubseteq} D$ and $C \dot{\equiv} D$. Let $\mathcal{N}_\mathcal{I}$ be a set of individual names (individuals for short); let $i, j \in \mathcal{N}_\mathcal{I}$. An *ABox* is a set of axioms (so-called ABox assertions) of the form $i : C$ (concept assertion) and $(i, j) : R$ (role assertion).

The *semantics* of a concept is given in terms of an interpretation. An *interpretation* is a pair $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a non-empty set (the interpretation domain), and $\cdot^\mathcal{I}$ is an interpretation function which maps concept names to subsets of $\Delta^\mathcal{I}$, roles to subsets of $\Delta^\mathcal{I} \times \Delta^\mathcal{I}$, and individuals to elements of $\Delta^\mathcal{I}$. Moreover, $\top^\mathcal{I} = \Delta^\mathcal{I}$ and $\bot^\mathcal{I} = \emptyset$ is required. The interpretation function can then be extended inductively to arbitrary concepts as follows:

- $(\neg C)^\mathcal{I} =_{def} \Delta^\mathcal{I} \setminus C^\mathcal{I}$,
- $(C \sqcap D)^\mathcal{I} =_{def} C^\mathcal{I} \cap D^\mathcal{I}$,
- $(C \sqcup D)^\mathcal{I} =_{def} C^\mathcal{I} \cup D^\mathcal{I}$,
- $(\exists R.D)^\mathcal{I} =_{def} \{\, i \in \Delta^\mathcal{I} \mid \exists j \in \Delta^\mathcal{I}.(i, j) \in R^\mathcal{I} \wedge j \in D^\mathcal{I} \,\}$, and
- $(\forall R.D)^\mathcal{I} =_{def} \{\, i \in \Delta^\mathcal{I} \mid \forall j \in \Delta^\mathcal{I}.(i, j) \in R^\mathcal{I} \rightarrow j \in D^\mathcal{I} \,\}$.

Please note that $C^\mathcal{I}$ is the extension of C. A concept C is satisfiable iff there exists an interpretation $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ such that $C^\mathcal{I} \neq \emptyset$; C *is satisfied* in $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ then. A TBox is satisfiable iff there is a $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ which satisfies all TBox axioms. A TBox axiom $C \dot{\sqsubseteq} D$ is satisfied by $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ iff $C^\mathcal{I} \subseteq D^\mathcal{I}$. A TBox axiom $C \dot{\equiv} D$ is satisfied by $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ iff $C^\mathcal{I} = D^\mathcal{I}$. An ABox is satisfiable iff there is an $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ which satisfies all ABox assertions. An ABox assertion $i : C$ is satisfied by $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ iff $i^\mathcal{I} \in C^\mathcal{I}$, and $(i, j) : R$ is satisfied by $(\Delta^\mathcal{I}, \cdot^\mathcal{I})$ iff $(i^\mathcal{I}, j^\mathcal{I}) \in R^\mathcal{I}$. To decide the consistency problem for C then means to decide whether there exists an interpretation which satisfies C, and analog for $\mathcal{T}$, $\mathcal{A}$. Often, one wants to decide the satisfiability problem for a whole *knowledge base* which is a pair $(\mathcal{T}, \mathcal{A})$. Tableau calculi can do this.

From a mathematical perspective, a tableau prover is non-deterministic calculus which applies a set of inference rules to an initial input ABox. The $\mathcal{ALC}$ tableau rules are shown in Fig. 1. Basically, there is

$\sqcap$-rule:
if 1. $i : C_1 \sqcap C_2 \in \mathcal{A}$
2. $\{i : C_1, i : C_2\} \not\subseteq \mathcal{A}$
then $\mathcal{A} := \mathcal{A} \cup \{i : C_1, i : C_2\}$

$\sqcup$-rule:
if 1. $i : C_1 \sqcup C_2 \in \mathcal{A}$
2. $\{i : C_1, i : C_2\} \cap \mathcal{A} = \emptyset$
then $\mathcal{A} := \mathcal{A} \cup \{i : D\}$
for some $D \in \{C_1, C_2\}$

$\forall$-rule:
if 1. $i : \forall R.D \in \mathcal{A}$
2. $(i, j) : R \in \mathcal{A}$
3. $j : D \notin \mathcal{A}$
then $\mathcal{A} := \mathcal{A} \cup \{j : D\}$

$\exists$-rule:
if 1. $i : \exists R.D \in \mathcal{A}$
2. $\{(i, j) : R, j : D\} \cap \mathcal{A} = \emptyset$ for all $j \in \mathcal{N}_\mathcal{I}$
then $\mathcal{A} := \mathcal{A} \cup \{(i, j) : R, j : D\}$ (for a new j)

Fig. 1. The $\mathcal{ALC}$ Tableau Expansion Rules

one tableau rule for each concept constructor. The input ABox becomes the working data structure of the calculus, the so-called tableau. The calculus attempts to construct a (finite representation) of a model of the input, witnessing satisfiability if successful. A tableau is simply an ABox which has been augmented by rule consequences. At runtime, each tableau rule checks its applicability on the current state of the tableau. In case no more rules can be applied, the tableau is said to be *complete*. The non-determinism works as follows: If the tableau rules can be applied *in such a way* that a complete and contradiction-free tableau can be derived, then the input ABox is satisfiable, and unsatisfiable otherwise. Please also note the non-determinism in the $\sqcup$-rule. A *contradictory tableau* $\mathcal{A}$ contains some subset $\{i : C, i : \neg C\} \subseteq \mathcal{A}$, for some i and C, a so-called *clash*.

For example, on the input ABox $\{betty : woman \sqcap \exists has\_child.person\}$ the completion $\{betty : woman \sqcap \exists has\_child.person, (betty, j) : has\_child, j : person\}$ will be constructed by the calculus. It has constructed a new individual j, representing the child of $betty$. Note that this tableau represents a model of the input ABox. The calculus can decide concept satisfiability, since C is satisfiable iff $\{i : C\}$ is.

4 Tableau Calculi from a Software Perspective

A non-deterministic calculus cannot be implemented directly on a deterministic computer. Thus, *search* is required to eliminate the non-determinism. From this perspective, each node in the search space corresponds to one possible tableau state which has been generated by the application of a tableau rule. Moreover, the tableau rules must be applied according to a certain *strategy* then in order to ensure formal properties of the calculus (like termination).

A very simple *concept satisfiability checker* which doesn't take the TBox into account in plain COMMON LISP (without MIDELORA) looks as follows. The prover neither uses optimizations, nor an *explicit tableau representation*. There is no graph-like data structure. Instead, the tableau is represented implicitly by the stack frames. The prover works on input concepts which are in negation normal form (NNF) only. This means that negation appears only in front of concept *names*; however, each concept can be brought into NNF. Concepts are in standard Lisp syntax, e.g., $woman \sqcap \exists has\_child.person \sqcap \forall has\_child.male$ is `(and woman (some has-child person) (all has-child male))`, also called KRSS syntax. A simple functional $\mathcal{ALC}$ prover looks as follows:

```
(defun alc-sat (concept)
  (labels ((alc-sat1 (expanded unexpanded)
             (labels ((get-negated-concept (concept)
                        (nnf '(not ,concept)))
                      (select-concept-if-present (type)
                        (find-if #'(lambda (concept)
                                     (and (consp concept)
                                          (eq (first concept) type)))
                                 unexpanded))
                      (select-atom-if-present ()
                        (find-if #'(lambda (concept)
                                     (or (symbolp concept)
                                         (and (consp concept)
                                              (eq (first concept) 'not)
                                              (symbolp (second concept)))))
                                 unexpanded))
                      (clash (concept)
                        (let ((negated-concept (get-negated-concept concept)))
                          (find negated-concept expanded :test #'equal)))
                      (register-as-expanded (concept)
                        (unless (clash concept)
                          (alc-sat1 (cons concept expanded)
                                    (remove concept unexpanded :test #'equal)))))
```

```
(let ((atom (select-atom-if-present)))
  (if atom
      (register-as-expanded atom)
    ;; else
    (let ((and-concept (select-concept-if-present 'and)))
      (if and-concept
          (progn
            (dolist (conjunct (rest and-concept))
              (when (clash conjunct)
                (return-from alc-sat1 nil))
              (push conjunct unexpanded))
            (register-as-expanded and-concept))
        ;; else
        (let ((or-concept (select-concept-if-present 'or)))
          (if or-concept
              (let ((unexpanded-old unexpanded))
                (some #'(lambda (arg)
                          (unless (clash arg)
                            (setf unexpanded
                              (cons arg unexpanded-old))
                            (register-as-expanded or-concept)))
                      (rest or-concept)))
            ;; else
            (let ((some-concept (select-concept-if-present 'some)))
              (if some-concept
                  (let* ((qualification (third some-concept))
                         (role (second some-concept))
                         (initial-label
                          (cons
                            qualification
                           (mapcar #'third
                               (remove-if-not
                                 #'(lambda (concept)
                                       (and (consp concept)
                                            (eq (first concept) 'all)
                                            (eq (second concept) role)))
                                   unexpanded)))))
                    (and (alc-sat1 nil initial-label)
                         (register-as-expanded some-concept)))
                ;; else
                t)))))))))

(alc-sat1 nil (list (nnf concept)))))
```

Some explanations are required. Basically, each incarnation of `alc-sat1` on the stack represents a state in the search space and thus also a tableau state. A rule can be seen as a generator which generates one or more successor search states. The latter is the case for the non-deterministic $\sqcup$-rule (`if or-concept ...`). Each search state has a list of `expanded` and `unexpanded` concepts. So, an assertion $i : C \in \mathcal{A}$ is reflected by C being a member of either the expanded or unexpanded list in the corresponding incarnation of `alc-sat1`. A rule application moves one or more concepts from unexpanded to expanded, and adds some other concepts to expanded. Expanded concepts are no longer considered by subsequent rule applications, until backtracking occurs, during which expanded concepts may become unexpanded again. This eliminates the need for checking the preconditions of the rules over and over again. The rule application strategy can easily be recognized in the code. Rules are applied in this order: $\sqcap, \sqcup, \exists$. The $\forall$-rule is integrated in the $\exists$-rule. Whenever a new R-successor j of i is created due to $i : \exists R.C \in \mathcal{A}$, the rule also takes care to collect all applicable D concepts, originating from $i : \forall R.D \in \mathcal{A}$ assertions, and then adds C and D to the unexpanded list of j.

This prover is very simple and cannot survive any complex input. Nevertheless, one can easily imagine that its source will become quite complex if state-of-the-art optimizations were included, such as dependency directed backtracking, semantic branching, and model merging. Moreover, an *explicit ABox representation* is required in order to implement an ABox consistency checker. In case this ABox representation is a boxed graph data structure (like in MIDELORA), also the problem of *backtracking* becomes apparent, since the tableau data structure will then be modified destructively during the tableau expansion, and the question of *how to revert the tableau data structure to a previous state during backtracking* arises. Obviously, one cannot simply work with tableau copies. MIDELORA uses undo operations for this purpose. It thus maintain a history of command objects [2, Command Pattern], recording the elementary tableau changes. A "roll back" is performed by traversing this history in reverse order, compensating the effects of the command objects one by one, until a desired goal state is reached (similar to the SAGAS known in transaction processing). An alternative approach is to use an explicit *and-or-graph representation.* We still need to evaluate whether an and-or-graph representation gives an acceptable performance.

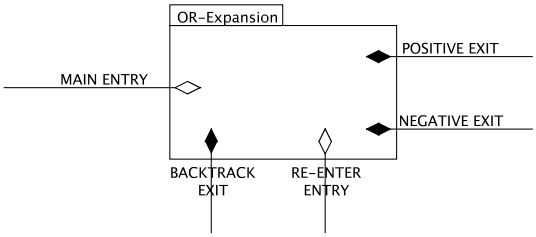

Fig. 2. 5-Port-Model of a MIDELORA Tableau Rule

5 A Tableau Prover in MIDELORA

Considering the rules as generators of successor states in the search space, the following Prolog-inspired 5-port model can be used to understand their behavior in the backtracking depth-first search algorithm. According to this 5-port model, each rule has the following ports, see Fig. 2. Remember that a rule incarnation on the stack correspond to a search state and thus to a tableau state also:

• if the search state is freshly created, the rule incarnation is created and entered using the *main entry.* The rule then checks its applicability on that tableau state.

• if the rule is not applicable, the *negative exit* is taken, and the calculation proceeds according to the global strategy which describes how the rules are wired / connected to one another (see below).

• if the rule is applicable, it modifies or creates a clash-free (new) successor state, and the *positive exit* is taken. The computation proceeds as described in the global strategy. If such a successor tableau cannot be created, backtracking occurs. Some other assertions in the tableau must be revised then; however, this can only be done by the corresponding rule incarnation which has added these assertions. With a dependency analysis, the corresponding state in the search space to backtrack to can be identified - the *backtracking destination*. The *backtrack exit* is taken, and control is passed to the parent incarnation in the search space. Moreover, information for identifying the backtracking destination is passed upwards via a `return`.

• If during backtracking a rule incarnation got back the control together with the information about the backtracking destination, then there are two possibilities. In case the rule incarnation is not the backtracking destination, the *backtrack exit* is taken, and backtracking continues. But if the rule incarnation is the backtracking destination, then it is partially responsible for the backtracking caused by the clash, and the *re-enter entry* is taken. The rule incarnation is thus asked to "revise its decision" and to create a different successor tableau, if possible. This request can only be fulfilled by non-deterministic rules. However, before an alternative tableau can be created, the state of the tableau must be reverted into its original state (this requires to roll back the command history in case a boxed tableau data structure is used). After that, the rule creates the next successor tableau, if possible. Otherwise, the rule has no more alternatives, so backtracking continues. Some other rule who *also* contributed to the contradiction has to revise its decision then (dependency information is exploited here again, but we cannot go into detail). The *backtrack exit* is taken, and the (revised) backtracking destination is passed upwards (`returned`).

The ports of the different rules can then be wired / connected together to create *rule chains*, implementing certain *global strategies.* Since the individual rules are highly optimized, the provers crafted in such a way exhibit a good performance. A simple $\mathcal{ALC}$ prover is shown in Fig. 3. The rule chain corresponds closely to the prover from Section 4. Wires which run completely inside the `ALC-SAT` box are within the same incarnation of `ALC-SAT` (this prover creates less stack frames than the prover in Section 4, since only non-deterministic rules have to spawn new incarnations to enable the backtracking). The `deterministic-expansion` includes the $\sqcap$-rule, but does some more things, e.g., takes care of a TBox and certain optimizations. However, in order to turn this prover into an ABox satisfiability checker, another "ABox preprocessor" prover has to be applied first, in the so-called *init phase*. This prover runs a different strategy – it *exhaustively* applies the rules (Fig. 1) with the exception of the $\exists$-rule (thus, no new individuals are created in this phase). Consequently, the init phase finishes when neither unexpanded conjunctions, open disjunctions nor non-applied universal role restrictions remain in the input ABox. Next, the *main phase* of the ABox satisfiability prover takes care to expand the remaining $\exists R.C$ assertions; this prover is basically identical to the prover from Section 4. Finally, if a completion has been found, the success can be signaled in the *success phase*.

The `defprover` macro is used to define a prover in the MIDELORA space:

```
(defprover ((<T> <DL> <S>) ...) (:init ...) (:main ...) (:success ...))
```

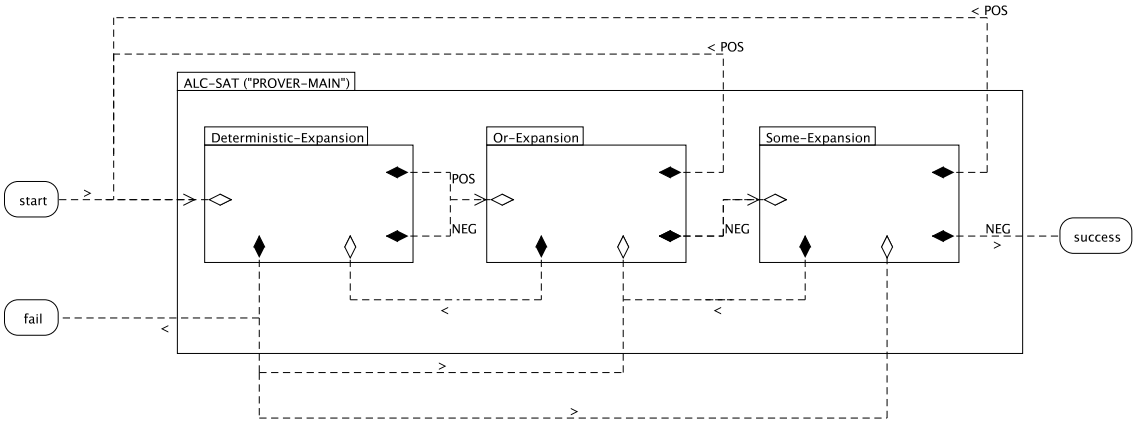

Fig. 3. MIDELORA Model of the $\mathcal{ALC}$ Prover

which defines a prover $(\texttt{<T>}, \texttt{<DL>}, \texttt{<S>}) \in \mathcal{T} \times \mathcal{L} \times \mathcal{S}$ in the MIDELORA space. A prover has three operational phases. In the `:init` phase, for example, the "ABox preprocessor" can be run. The `:main` phase does the main work. Finally, there is the `:success` phase. The macro sets up three ternary CLOS methods `prover-init, prover-main, prover-success`, which dispatch on `<T>, <DL>, <S>`. However, as explained, dispatch for the `<DL>` argument works in a non-standard way.

A complete $\mathcal{ALC}$ ABox consistency checker then looks as follows. Note that this prover is defined for substrates of type `abox` for the language `alc`, and it solves the `abox-sat` task:

```
(defprover ((abox-sat alc abox))
  (:init
   (perform (initial-abox-saturation)
     (:body
      (start-main))))
  (:main
      (perform (deterministic-expansion)
        (:body
         (if clashes
             (handle-clashes)
           (perform (or-expansion)
             (:positive
              (if clashes
                  (handle-clashes)
                (restart-main)))
             (:negative
                 (perform (some-expansion)
                   (:positive
                    (if clashes
                        (handle-clashes)
                      (restart-main)))
                   (:negative
                    (success)))))))))
  (:success
   (completion-found)))
```

We claim that this is a very comprehensible description of an $\mathcal{ALC}$ ABox consistency checker. Of course, the main code is in the definitions of the rules, which are highly optimized. As long as no additional parameters for adaptation or configuration of the rules are required, all parameters are passed and handled implicitly within these macros ("macro variable capture"). The rule bodies must then obey certain conventions. We believe that abstracting from irrelevant details such as parameter names and method / macro signatures can help the developer to focus on the more important and intellectually more demanding details during implementation. The `define-rule` macro defines a rule for a pair $(\texttt{<DL>}, \texttt{<S>}) \in \mathcal{L} \times \mathcal{S}$ in the MIDELORA space. The following $\exists$-rule is defined for all "DLs with somes" (a mixin class):

```
(defrule some-expansion (dl-with-somes abox)
  (multiple-value-bind (some-concept node)
      (select-some-concept abox *strategy* language)
    (cond ((not node)
           +insert-negative-code+ )
          (t
           (let ((role (role some-concept))
                 (new-node nil))
             (register-as-expanded some-concept :node node)
```

```
(setf new-node
      (create-anonymous-node abox
       :depends-on (list (list node some-concept))))
(relate node new-node role
        :old-p nil
        :depends-on (list (list node some-concept)))
(perform (compute-new-some-successor-label
               :new-node new-node
               :node node :role role :concept some-concept))
+insert-positive-code+ )))))
```

The markers `+insert-positive-code+` and `+insert-negative-code+` are replaced by the appropriate `:positive` and `:negative` S-expressions specified in the rule strategy. The compiler can either replace the markers literally with the `:positive, :negative` S-expression (macro expansion), or code a function call here. Please note that `compute-new-some-successor-label` works as described in Section 4. The access to the substrate data structure via dedicated methods is shown. All changes to the tableau are automatically recorded in the history in order to enable the roll back during backtracking.

Sometimes, a rule written for a DL $\mathcal{DL}'$ can in principle inherit the implementation of a rule defined for another DL, $\mathcal{DL}$. In case the language class for $\mathcal{DL}'$ is not a subclass of the language class for $\mathcal{DL}$, the explicit reuse via `:inherit-from` can be enforced: `(defrule or-expansion (alc abox) :inherit-from (alchi abox) ...)` Rules, in principle, are not limited to tableau rules. Rules which manage the memory, or implement dedicated optimizations techniques (such as model caching and model merging) can be defined and interwoven easily into a prover / rule chain. In fact, a highly optimized $\mathcal{ALC}$ prover contains some more rule applications in its strategy, such as `cache-and-delete`, `model-merging`, etc. But this is not exemplified here. Even with these optimizations in the chain, the prover is still very comprehensible and concise.

6 Conclusion

We have introduced the basics of description logics and the dominant family of reasoning calculi for these logics - tableau algorithms. Although tableau algorithms are very concise, their naive implementation can result in monsters of software complexity. To alleviate these problems and to offer a flexible platform for DL system implementation, the MIDELORA framework and its software abstractions were introduced. We have exemplified their benefit. COMMON LISP provides an ideal platform for our endeavor. Features sometimes considered as "esoteric" or too complicated by users of simpler programming languages turned out to be very valuable or even indispensable here, e.g., macros, multi-methods, multiple inheritance, and custom method combinations (e.g., to realize contra-variant dispatch for certain method arguments).

One could argue that our software architecture has a very high memory footprint, and thus will also not perform very well. The former is, in principle, true. Since "everything is an object" (substrates, nodes, edges, ...), and dynamic multi-dispatch is used at many places, one might be skeptical whether provers defined in that way will actually be able to perform very well (with or without optimizations). Moreover, the use of an object-oriented boxed data structures implies that a command history documenting the tableau changes must be maintained, in order to be able to roll back the tableau into a previous state during backtracking. These histories can become very long as well. However, benchmarks have shown that the resulting provers can still run reasonably fast [4], but do not yet perform as well, as, say, RACERPRO. But techniques to reduce the memory footprint and thus also to save garbage collection time are well known (pooled data structures, flyweight and/or proxy objects [2]). Mature COMMON LISP implementations also offer ways to tune and configure the garbage collector. We admit that some more work is required here. Finally, we like to observe that there seems to be a close affinity with work carried out in *software product families.* We would also like to thank the anonymous reviewers for valuable comments and suggestions.

References

1. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook – Theory, Implemenation and Applications.* Cambridge University Press, 2003.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
3. R. Möller, V. Haarslev, and M. Wessel. On the Scalability of Description Logic Instance Retrieval. In *29. Deutsche Jahrestagung für Künstliche Intelligenz (KI '06)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2006.
4. M. Wessel. Flexible und konfigurierbare Softwarearchitekturen für ontologiebasierte Informationssysteme. PhD thesis, Hamburg University of Technology, 2007.
5. M. Wessel and R. Möller. Flexible Software Architectures for Ontology-Based Information Systems. *Journal of Applied Logic – Special Issue on Emperically Sucessful Systems*, 2007.