Using Non-standard Inferences in Description Logics—what does it buy me? *

Sebastian Brandt and Anni-Yasmin Turhan,
Theoretical Computer Science,
RWTH Aachen, Germany
Email: {sbrandt, turhan}@cs.rwth-aachen.de

Abstract

In knowledge representation systems based on Description Logics, standard inference services such as consistency, subsumption, and instance are well-investigated. In contrast, non-standard inferences like most specific concept, least common subsumer, unification, and matching are missing in most systems—or exist only as ad-hoc implementations. We give an example of how these inferences can be applied successfully in the domain of process engineering. The benefit gained in our example, however, occurs in to many domains where knowledge bases are managed by persons with little expertise in knowledge engineering.

1 Process Engineering

As an application domain for knowledge representation systems based on Description Logics (DL-systems) in general, and certain non-standard inferences in particular, we give a brief introduction to the basic notions of the field of process engineering. In this context, a *process* is defined as a sequence of physical, chemical, biological, and informational operations intentionally executed to change substances in respect to their nature, properties, and composition.

Process engineering is concerned with methods, tools, and their management for the design and control of a process.[1] Here, *models* are used to represent, analyze, and optimize processes and get a deeper understanding of their nature. In general, a model is an abstraction of some object under consideration characterized by a lower level of complexity while retaining some of the original properties of interest.

*This work has been supported by the DFG, Project BA 1122/4-1.

[1] *Plant engineering* in turn deals with the actual (chemical) plant performing the process and its construction, which is abstracted from in process engineering.

In process engineering, exact equation-based mathematical models are particularly desirable because of their high predictive capabilities in numerical analysis and simulation. Unfortunately, even for simple chemical processes, such models are too complex for ad-hoc construction by hand. Nevertheless, adequate models can be obtained step by step, starting with other representation formalisms, e.g., so-called *block-oriented* models. In such models, a process is represented by an undirected graph with blocks as vertices and connections as edges. Each *block* stands for a standardized sub-unit of the entire process with certain interfaces and each *connection* for a flow of material, energy, or information. The type of a connection linking two interfaces of blocks is determined by the interface specifications. Typically, block-oriented modeling environments have a *block repository* in which building blocks are stored.

During the life-cycle of a chemical process, several models on different levels of detail are involved. In an early design stage, for instance, rather crude models allow to consider alternative designs in minimal time. Once one of them proved promising, more accurate models are used for further examination. With such a cascade of models, however, it is not clear how to benefit from one modeling stage when going into further detail on the next.

In answer to this, several requirements have been identified for block-oriented models and appropriate modelling environments in process engineering [15]:

- *Variable granularity*: The model should allow composite building blocks, i.e., blocks again comprising blocks and connections. These can be decomposed during the design phase until the desired level of detail is reached.

- *Generic building blocks*: A block in the repository should not be fully specified but rather represent a class of some subunit. During the design procedure, particular instances are obtained by specifying the relevant variables, equations, and values abstracted from in the classes.

- *Structured storage*: To avoid unnecessary extensions of the block repository and to facilitate browsing and searching, the existing blocks should be arranged in an "is-specialization-of" hierarchy.

- *Automatic classification*: If the specialization order would be derivable automatically, the system could additionally maintain consistency of new building blocks during the design procedure and locate the correct positions for their storage in the repository.

- *Re-use of submodels*: It should be possible to store (abstractions of) subunits in existing models in the repository for later re-use.

- *Maintenance support*: As the block repository typically will be developed over a long period of time by many people, detecting redundancies and integrating additional repositories should be possible.

The challenge to meet these requirements has inspired a cooperation between the Institute for Process Systems Engineering at RWTH Aachen, where a prototype modelling environment is being developed, and our research group, where DL-systems are studied. It has already been shown that DL-systems can successfully be employed for most of the above tasks [15]. Testing the developed prototype environment has provided additional insights. When designing models by means of block-oriented modelling environments, process engineers showed two characteristic strategies for the design of new (generic) blocks:

- *Bottom-up design*: From several existing process models, the process engineer selects a certain collection of subunits deployed for a similar purpose. She then introduces a new generic block as an abstraction of these units.
- *Design by modification*: Before assembling a new generic block from scratch, the knowledge engineer tries to locate a structurally similar one in the repository. She then modifies the existing block to suit the new requirements.

In this work, we will show how these design strategies can be supported by non-standard inferences offered by DL-systems. In the following section, we will introduce Description Logics formally and discuss their benefit for the requirements mentioned above. Sections 3 and 4 describe the particular non-standard inference services used to support the two design techniques.

2 Description Logics and Process Modelling

Description Logics (DL) form a category of knowledge representation (KR) formalisms used to represent terminological knowledge in a structured and well-defined way. A DL-system consists of a *knowledge base* together with certain *inference services*. The knowledge base comprises two components, the *TBox* and the *ABox*. Intuitively, the TBox defines the vocabulary by which a concrete world (in this application a process model) is described in the ABox. Both are defined by means of *concepts*, whose syntax and semantics is introduced next.

Concepts are inductively defined using a set of concept *constructors*, starting from a set N_C of *concept names* and a set N_R of *role names*. The constructs available in the DLs considered here are listed in Table 1. In $\mathcal{EL}$, the top-concept ($\top$), conjunction ($C \sqcap D$), and existential restriction ($\exists r.C$) are allowed. $\mathcal{ALE}$ additionally provides the bottom-concept ($\bot$) and primitive negation ($\neg A$). $\mathcal{ALN}$ extends $\mathcal{ALE}$ with number restrictions ($\geq n\, r$) and ($\leq n\, r$), but does not provide existential restrictions. A concept defined over the DL $\mathcal{L}$ ($\mathcal{L} \in \{\mathcal{EL}, \mathcal{ALE}, \mathcal{ALN}\}$) is referred to as $\mathcal{L}$-concept.

The semantics of concepts is defined in terms of an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. The *domain* $\Delta^{\mathcal{I}}$ is a non-empty set, and the *interpretation function* $\cdot^{\mathcal{I}}$

Syntax	Semantics	$\mathcal{EL}$	$\mathcal{ALE}$	$\mathcal{ALN}$
$\top$	$\Delta^{\mathcal{I}}$	x	x	x
$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	x	x	x
$\forall r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y\colon (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$		x	x
$\exists r.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y\colon (x,y) \in r^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$	x	x	
$\bot$	$\emptyset$		x	x
$\neg A$, $A \in N_C$	$\Delta^{\mathcal{I}} \setminus A^{\mathcal{I}}$		x	x
$(\geq n\, r)$, $n \in \mathbb{N}$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \mid (x,y) \in r^{\mathcal{I}}\} \geq n\}$			x
$(\leq n\, r)$, $n \in \mathbb{N}$	$\{x \in \Delta^{\mathcal{I}} \mid \#\{y \mid (x,y) \in r^{\mathcal{I}}\} \leq n\}$			x

Table 1: Syntax and semantics of concepts.

maps every concept name $A \in N_C$ to a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and each role name $r \in N_R$ to a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The second column of Table 1 shows how $\cdot^{\mathcal{I}}$ is extended to complex concepts.

Definition 1 (TBox) *A TBox $\mathcal{T}$ is a finite set of* concept definitions *of the form $A \doteq C$, where $A \in N_C$ and C is a concept. Every concept name A may occur only once on a left-hand side in $\mathcal{T}$. If it does, then A is called* defined*, otherwise* primitive*. In DLs providing primitive negation only primitive concepts may be negated on the right-hand side of concept definitions. An interpretation $\mathcal{I}$ is a* model *for $\mathcal{T}$ iff $A^{\mathcal{I}} = C^{\mathcal{I}}$ for every $A \doteq C \in \mathcal{T}$.*

To illustrate the introduced notions of concept, concept definition and TBox, consider an example TBox.

Example 2 *The $\mathcal{ALE}$-TBox $\mathcal{T}_{ex}$ contains the following concept definitions inspired by the process engineering domain:*

$$
\begin{aligned}
\mathsf{Liquid} &\doteq \neg\mathsf{Solid} \sqcap \neg\mathsf{Gas},\\
\mathsf{Container} &\doteq \mathsf{Volume} \sqcap (\forall\, \mathsf{contains}.\, \mathsf{Substance}),\\
\mathsf{FluidTank} &\doteq \mathsf{Container} \sqcap (\forall\, \mathsf{hasConnection}.\, \mathsf{Port}) \sqcap\\
&\quad\ (\exists\, \mathsf{contains}.\, \mathsf{Liquid}),\\
\mathsf{Pipeline} &\doteq \mathsf{Volume} \sqcap \mathsf{Tube}\\
&\quad\ (\forall\, \mathsf{hasConnection}.\, (\mathsf{Port} \sqcap (\exists\, \mathsf{hasPart}.\, \mathsf{Valve}))) \sqcap\\
&\quad\ (\forall\, \mathsf{contains}.\, \mathsf{Substance}) \sqcap (\exists\, \mathsf{contains}.\neg\mathsf{Solid})
\end{aligned}
$$

In the TBox $\mathcal{T}_{ex}$ the concept Liquid *is defined as something that is no* Gas *and no* Solid*. A* Container *is defined as a* Volume *containing only* Substances*. Based on these two defined concepts, a* FluidTank *is defined as a* Container *which contains a* Liquid *and is only connected to* Ports*. Finally, a* Pipeline *is defined as a* Volume *and a* Tube *and is only connected to* Ports *which in turn must have a* Valve *as a part. Furthermore, a* Pipeline *must contain something, which is no* Solid *and all it contains are* Substances*.*

To represent knowledge about an actual instance of the application domain, individuals and their interrelations are described in an ABox. Thus, in addition to N_C and N_R, we introduce a finite set N_I of *individual names*. Formally, an ABox can now be defined as follows:

Definition 3 (ABox) *An ABox $\mathcal{A}$ is a finite set of* concept assertions *of the form* $\mathsf{a}\colon C$ *and* role assertions *of the form* $(\mathsf{b},\mathsf{c})\colon r$, *where* $\mathsf{a},\mathsf{b},\mathsf{c} \in N_I$, C *is an arbitrary concept, and* $r \in N_R$ *a role name. An interpretation $\mathcal{I}$ is a* model *for $\mathcal{A}$, iff* $\mathsf{a}^{\mathcal{I}} \in C^{\mathcal{I}}$ *and* $(\mathsf{b}^{\mathcal{I}},\mathsf{c}^{\mathcal{I}}) \in r^{\mathcal{I}}$ *for every* $\mathsf{a}\colon C$ *and every* $(\mathsf{b},\mathsf{c})\colon r$ *in $\mathcal{A}$. For every interpretation $\mathcal{I}$, every* $\mathsf{a} \in N_I$ *is mapped to some* $\mathsf{a}^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, *such that* $\mathsf{a} \neq b$ *implies* $\mathsf{a}^{\mathcal{I}} \neq b^{\mathcal{I}}$ *(unique name assumption).*

In our process engineering application, each of the individual blocks is represented by an individual in an ABox. The generic blocks from the repository are represented by concepts defined in a TBox. Thus, TBox and ABox form a knowledge base for all blocks constructed in the modelling environment as illustrated in Figure 1.

To derive implicit knowledge from the explicit one given in the knowledge base, there are three so-called *standard inferences*, namely *consistency*, *subsumption*, and *instance*, as defined below.

Definition 4 (Standard inferences) *A concept C is* consistent *iff there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$. A concept C is* subsumed *by a concept D (written $C \sqsubseteq D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for all interpretations $\mathcal{I}$. The concepts C and D are* equivalent *(written $C \equiv D$) iff they subsume each other. An individual name $a \in N_I$ is an* instance *of C w.r.t. an ABox $\mathcal{A}$ and its TBox $\mathcal{T}$ (written $\mathsf{a} \in_{\mathcal{A},\mathcal{T}} C$) iff $\mathsf{a}^{\mathcal{I}} \in C^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{A}$ and $\mathcal{T}$.*

These inferences are essential for almost all DL-systems. Especially, computing the so-called subsumption hierarchy of concepts yields the "is-specialization-of"-hierarchy mentioned in Section 1. Algorithms deciding subsumption form the basis for structured storage and the algorithms for computing the "instance-of"-relation realize the automatic classification of objects. Not all of the tasks mentioned in Section 1 can be accomplished by means of standard inferences, e.g., they do not facilitate the previously mentioned design strategies utilized by process engineers. This is where the *non-standard inferences* come into play.

3 Supporting the Bottom-up Approach

The bottom-up generation of a new block (i.e., concept) from a set of process models (i.e., ABox individuals) selected by the domain expert is realized by non-standard inferences in two steps. Firstly, the most specific concept is computed for each of the selected ABox individuals, such that the individual is an instance

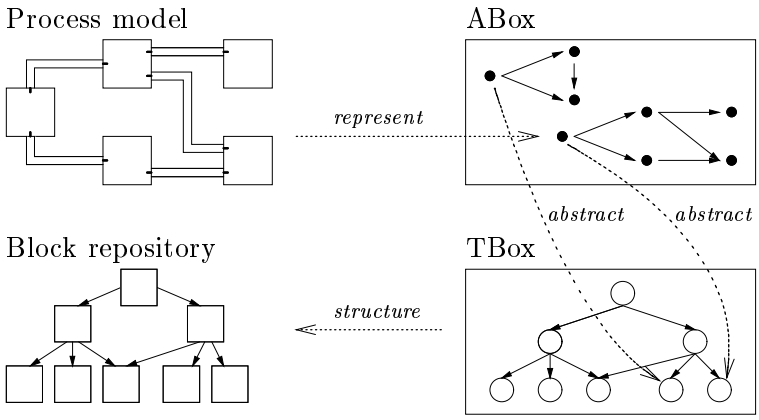

Figure 1: Modelling environment and knowledge base

of the obtained concept which is most specific w.r.t. subsumption. Next, a single concept is computed from all the obtained concepts, which subsumes all the obtained concepts and is also the most specific concept to do so. The resulting concept is then offered to the process engineer for inspection and further processing and, if suitable, added to the generic block repository. The first step is realized by the non-standard inference *most specific concept* (msc) which is defined in the following way:

Definition 5 (msc) *Let $\mathcal{A}$ be an $\mathcal{L}$-ABox,* a *an individual in $\mathcal{A}$ and C a concept in $\mathcal{L}$, then C is the* most specific concept (msc) *of* a *w.r.t. $\mathcal{A}$ ($\mathsf{msc}_{\mathcal{A}}(\mathsf{a})$) iff* $\mathsf{a} \in_{\mathcal{A}} C$, *and for all $\mathcal{L}$-concepts C',* $\mathsf{a} \in_{\mathcal{A}} C'$ *implies* $C \sqsubseteq C'$.

Computing the msc of an individual yields an abstraction from a concrete individual and from its interrelationships expressed in the ABox by generalizing it into a concept.

Example 6 *As an example inspired from the application domain, we want to describe a distillation device which takes sea-water as an input and separates it into water and salt. Such a device could be represented by an $\mathcal{EL}$-ABox $\mathcal{A}_{ex}$ with the following assertions:*

$$\begin{array}{rl} \mathsf{device}: & \mathsf{MarineDistiller}, \\ (\mathsf{device}, \mathsf{seawater}): & \mathsf{hasInput}, \\ (\mathsf{device}, \mathsf{water}): & \mathsf{hasOutput}, \\ (\mathsf{device}, \mathsf{salt}): & \mathsf{hasOutput}, \end{array} \qquad \begin{array}{rl} \mathsf{seawater}: & \mathsf{solution} \sqcap \mathsf{Liquid}, \\ (\mathsf{seawater}, \mathsf{water}): & \mathsf{contains}, \\ (\mathsf{seawater}, \mathsf{salt}): & \mathsf{contains}, \\ \mathsf{water}: & \mathsf{Solvent} \sqcap \mathsf{Liquid}, \\ \mathsf{salt}: & \mathsf{Solute} \sqcap \mathsf{Solid} \end{array}$$

Note that the individuals water *and* salt *occur as role-successors for both of the individuals* device *and* seawater. *The* msc(device) *w.r.t. the underlying ABox $\mathcal{A}_{ex}$ is now given by:*

$$\begin{aligned}\mathsf{msc}_{\mathcal{A}_{ex}}(\mathsf{device}) = \;& \mathsf{MarineDistiller} \sqcap \\ & \exists\, \mathsf{hasInput}.\,(\mathsf{Solution} \sqcap \mathsf{Liquid} \sqcap \\ & \qquad \exists\, \mathsf{contains}.\,(\mathsf{Solvent} \sqcap \mathsf{Liquid}) \sqcap \\ & \qquad \exists\, \mathsf{contains}.\,(\mathsf{Solute} \sqcap \mathsf{Solid})) \sqcap \\ & \exists\, \mathsf{hasOutput}.\,(\mathsf{Solvent} \sqcap \mathsf{Liquid}) \sqcap \\ & \exists\, \mathsf{hasOutput}.\,(\mathsf{Solute} \sqcap \mathsf{Solid})\end{aligned}$$

In the obtained concept the concept names from the ABox $\mathcal{A}_{ex}$ are preserved in the msc *concept and the interrelations are expressed by existential restrictions. The co-references in $\mathcal{A}_{ex}$ to each of* water *and to* salt *can not be captured in the concept, instead the concepts from $\mathcal{A}_{ex}$ corresponding to these individuals are duplicated.*

Unfortunately, the msc need not always exist due to cyclic relationships between ABox individuals such as $\{(\mathsf{a},\mathsf{b})\colon r, (\mathsf{b},\mathsf{a})\colon r\} \subseteq \mathcal{A}$. An individual from a cyclic ABox may be instance of all concepts from an infinite sequence of concepts C_1, C_2, ... where each concept C_i encodes one more traversal of the cycle expressed in the ABox than C_{i-1} and is thereby more specific than all its predecessors in the infinite sequence of concepts. Since the individual is an instance of *all* C_is, the most-specific concept would be $\sqcap_{i=1}^{\infty} C_i$, which cannot be expressed in every DL with existential restrictions. For further details, refer to [2].

However, for DLs providing existential restrictions the msc for cyclic ABoxes can be approximated by the so-called *k-approximation*. The k-approximation is a msc whose nesting depth of quantifiers is bounded by k ($k \in \mathbb{N}$). See [12] for details. Once in our process engineering application the k-approximation or, if possible, the msc of each selected individual block is attained, the subsuming concept—the least common subsumer—of them is computed. It is defined as follows:

Definition 7 (lcs) *Let $\mathcal{T}$ be an $\mathcal{L}$-concept and C_1, ..., C_n concepts in $\mathcal{L}$ from $\mathcal{T}$, then C is the* least common subsumer (lcs) *of C_1, ..., C_n w.r.t. $\mathcal{T}$ ($\mathsf{lcs}_{\mathcal{T}}(C_1, \ldots, C_n)$) iff $C_i \sqsubseteq_{\mathcal{T}} C$ for all $1 \leq i \leq n$, and for all $\mathcal{L}$-concepts C', $C_i \sqsubseteq_{\mathcal{T}} C'$ for all $1 \leq i \leq n$ implies $C \sqsubseteq C'$.*

Thus, both the msc and the lcs generalize the input yielding the most specific concept w.r.t. the underlying TBox; only that the msc refers to a single ABox individual while the lcs refers to several concepts based on conccepts defined in a TBox.

Example 8 *Let us consider the* lcs *of the concepts* FluidTank *and* Pipeline *as defined in the TBox $\mathcal{T}_{ex}$ from Example 2. First, both concepts have to be unfolded w.r.t. $\mathcal{T}_{ex}$, i.e., all names of defined conepts are replaced recursively by the right-hand sides of their concept definitions. Next, the* lcs *concept of both input concepts is computed. In this case we obtain:*

$\mathsf{lcs}_{\mathcal{T}_{ex}}$ (FluidTank, Pipeline) =
Volume $\sqcap$ ($\forall$ hasConnection. Port) $\sqcap$
($\forall$ contains. Substance) $\sqcap$ ($\exists$ contains. (Substance $\sqcap$ $\neg$ Solid)).

The lcs *concept reflects that both input concepts are a* Volume*, because* Volume *lies in the intersection of the concept names on top-level of both input concepts. For the concept occurring in value restrictions, the* lcs *algorithm is applied recursively for each role. The existential restrictions in the* lcs *concept are obtained by conjoining the concepts in the existential restriction and those in their corresponding value restriction for each of the input concepts and then recursively applying the* lcs *algorithm to the obtained conjunctions.*

For the DLs introduced in Section 2, the lcs always exists. The lcs (as well as the msc, if it exists) is uniquely determined up to equivalence. In our research group, algorithms for the lcs have been developed for several DLs, see [10, 5, 2, 11].

Equipped with the non-standard inferences msc and the lcs, the demand from our application domain to construct knowledge bases in a bottom-up fashion can be met. The knowledge engineer selects some fully specified blocks that should form the new generic block for the repository. Then the blocks are automatically translated into individuals in an ABox, representing the parts and properties of each of the blocks. Next, the DL-system computes the msc of each of them and then generalizes them into a single concept by computing the lcs. The resulting concept is then translated back into the representation used in the modelling environment and offered as a new block to the process engineer, see Figure 1. Note that the domain expert is not involved in the "DL-part" of this process, therefore our method is suitable for users with little KR expertise.

In our application, the lcs inference does not only support the bottom-up approach for augmenting the repository. It may in addition be employed to obtain a well-structured storage in the repository which in turn is necessary for easily retrieving generic blocks for a possible re-use. So, if a generic block in the repository has many specializations, say $B_1, \ldots, B_n$ for a large number n, and the process engineer searches for a building block to re-use, inspecting all of the B_is to find a candidate may not be practical. New generic blocks subsuming some of the B_is and thereby providing an intermediate level in the specialization hierarchy facilitates browsing it. Such intermediate blocks can be derived by computing the least common subsumers of some of the B_is and adding them to the repository.

The lcs w.r.t. TBoxes has been implemented for the DL $\mathcal{ALE}$. As seen in Example 8, all input concepts have to be unfolded completely against the underlying TBox before computing the actual lcs concept. It is well-known that unfolding a concept can cause an exponential blow-up of the concept size [14].

Therefore, the concepts to be handled and—even worse—returned by the lcs algorithm can become very large. This does not only slow down the computation of the lcs, but also yields unreadable concepts. First empirical evaluations of our lcs-implementation applied to TBoxes from the process engineering domain have shown that the returned concepts fill several pages of output and are therefore too big to be readable and comprehensible for a human reader, see [7, 16].

Thus, for assessment by domain experts, the resulting concepts have to be represented more compactly. To this end, our research group investigates methods for finding a *minimal rewriting* of a concept w.r.t. the underlying TBox [6]. In a minimal rewriting, parts of the concept are replaced by names defined in the TBox. The effect of such a rewriting is somewhat inverse to unfolding, e.g., in Example 8 the lcs can be represented in a more compact way by using the definition in $\mathcal{T}_{ex}$ and replacing the sub-term of the lcs concept "$\mathsf{Volume} \sqcap (\forall\, \mathsf{contains}.\, \mathsf{Substance})$" by "$\mathsf{Container}$".

For DLs with existential restrictions, the computations of a minimal rewriting involves a high degree of non-determinism. Therefore, we have to resort to heuristics yielding small but not always minimal rewritings. In the case of the TBox used in our process engineering application, for instance, employing such heuristics to concepts of size 800 yields concepts of size 10. Refer to [7, 16] for details.

Moreover, rewritings can be used to "translate" concepts from one DL $\mathcal{L}_1$ into concepts from another, less expressive DL $\mathcal{L}_2$ by computing the best *approximation* of the concept. This service is especially desirable if more inference services are available in $\mathcal{L}_2$.

4 Supporting the Modification Approach

Another useful non-standard inference is matching, which was first proposed in the DL-system CLASSIC [13, 9]. In order to define matching, we need to introduce *concept patterns*.

Let $\mathcal{L}$ be any of the DLs introduced in Section 2 together with the sets N_C, N_R, and N_I. Additionally, let N_X be a finite set of *concept variables* disjoint to $N_C \cup N_R \cup N_I$. *$\mathcal{L}$-concept patterns* are $\mathcal{L}$-concepts for which in addition concept variables can be used in the place of concept names—except for the fact that the primitive negation ($\neg$) may not occur in front of variables. A *substitution* σ is a mapping from N_X into the set of $\mathcal{L}$-concepts. It is extended to concept patterns P by replacing every occurrence of $X \in N_X$ in P by $\sigma(X)$. Thus, $\sigma(P)$ again is an $\mathcal{L}$-concept. With these preliminaries we can define matching problems as follows:

Definition 9 (matching problems) *An* $\mathcal{L}$-matching problem *is of the form* $C \equiv^? P$, *where* C *is an* $\mathcal{L}$*-concept and* P *an* $\mathcal{L}$*-concept pattern. A substitution*

σ is a matcher *for $C \equiv^? P$ iff $C \equiv \sigma(P)$, i.e., σ replaces the variables in P by concepts in such a way that equivalence holds.*

As a trivial example, consider the matching problem $\mathsf{A} \sqcap \forall \mathsf{r}.\mathsf{B} \equiv^? X \sqcap \forall \mathsf{r}.Y$. An appropriate matcher would be, for instance, $\{X \mapsto \mathsf{A} \sqcap \forall \mathsf{r}.\mathsf{B}, Y \mapsto \mathsf{B}\}$.

Intuitively, if a concept can be matched against a pattern P, then their syntax trees share the "upper part", i.e., where P is fully specified, while deviations may occur at leaves labelled with variables. Hence, the set of all concepts that can be matched against P contains infinitely many concepts structurally similar to P to some extent. In this sense, matching P against several concepts and returning those which can be matched, can be seen as a search with the fully specified part of P as search criterion.

Let us return to the process engineer designing a new generic block in a block-oriented modelling environment as described in Section 1. For the strategy of design by modification, the crucial step is to find a generic block in the repository *structurally* similar to what the knowledge engineer intends to design. As an example, assume a modelling task for a fluid tank equipped with a cooling system and an equivalent backup cooling system, each with a thermostat controller. A convenient starting point for the design could be a block comprising a fluid tank combined with an arbitrary device controlled by some control units and a similar backup device.

Example 10 *With a matching algorithm at hand, relevant generic blocks could be found by matching the following concept pattern against every concept in the KB.*

$$\exists \mathsf{hasPart}.(\mathsf{FluidTank} \sqcap \exists \mathsf{hasPart}.(\neg \mathsf{BackupDevice} \sqcap X \sqcap \exists \mathsf{hasPart}.\mathsf{Controller}) \sqcap \exists \mathsf{hasPart}.(\mathsf{BackupDevice} \sqcap X \sqcap \exists \mathsf{hasPart}.\mathsf{Controller}))$$

The pattern specifies blocks consisting of at least one fluid tank equipped with at least two equivalent components one of which is a backup device and one not. Both components must have a controller.

The query could thus also return blocks with two or more tanks or, for instance, with duplicate heating systems or stirring devices. Nevertheless, by additionally retrieving a block representing a single cooling device and another for a single thermostat control unit, the engineer is well prepared to complete the design task efficiently. Naturally, usability issues suggest to hide the formal construction of patterns by user-friendly query front-ends.

Note that in all admissible concepts, both occurrences of X must be replaced by the same concept. This structural constraint cannot be expressed by simple "wildcards" familiar from ordinary search engines.

Formal means exist for further refinement of such pattern-driven searches. For the DL $\mathcal{ALN}$, so-called *side conditions* have been proposed to restrict the concepts a variable may be replaced with [1]. In the above example, we may want to restrict the query to duplicate devices with, say, temperature-related functionality. To this end, we could use a side condition of the form

$$X \sqsubseteq^{?} \mathsf{ThermalDevice},$$

thus including only those devices represented by the concept ThermalDevice.

Apart from supporting the technique of design by modification, matching can also help to provide maintenance support for the block repository, as described in Section 1 [8]. Matching algorithms for deciding and solving matching problems have been proposed in the DLs $\mathcal{ALE}$ and $\mathcal{ALN}$ [3, 4]. In $\mathcal{ALE}$, the decision problem is NP-complete, whereas the computation problem is EXPTIME-complete. In $\mathcal{ALN}$, the decision and computation problem are polynomial.

5 Conclusion and Future Work

We have presented an application where non-standard inference services can significantly enhance the usability of DL-systems. Here these services were proposed to assist process engineers in their practical techniques of designing process models. These techniques, however, are not specific to this very domain but apply to any scenario where knowledge bases are managed by domain experts with little expertise in knowledge representation.

With the DLs presented here, not all properties of the described models can be represented sufficiently. The demand for more expressive DLs, however, also necessitates to adapt the existing inference services to new language constructors such as qualified number restrictions, transitive roles, and role hierarchies.

Approaches to capture the relevant extensions by appropriate algorithms for non-standard inferences are currently studied by our research group. Additional language constructors can further increase the computational complexity of such algorithms. Nevertheless, experience has shown that a high worst-case complexity often can be tolerated as long as a moderate average-case complexity is observed in practical applications.

A promising alternative might be to realize non-standard inferences for expressive description logics by approximating the input concepts in a less expressive DL, where the desired inferences can be realized more efficiently.

References

[1] F. Baader, S. Brandt, and R. Küsters. Matching under side conditions in description logics In B. Nebel, editor, *Proc. of IJCAI-01*, 2001.

[2] F. Baader and R. Küsters. Computing the least common subsumer and the most specific concept in the presence of cyclic $\mathcal{ALN}$-concept descriptions. In O. Herzog and A. Günter, editors, *KI-98*, volume 1504 of *Lecture Notes in Computer Science*, pages 129–140, Bremen, Germany, 1998. Springer-Verlag.

[3] F. Baader and R. Küsters. Matching in description logics with existential restrictions. In *Proc. of KR2000*, pp. 261–272, Morgan Kaufmann Publishers, 2000.

[4] F. Baader, R. Küsters, A. Borgida, and D. McGuinness. Matching in description logics. *Journal of Logic and Computation*, 9(3):411–447, 1999.

[5] F. Baader, R. Küsters, and R. Molitor. Computing least common subsumer in description logics with existential restrictions. In T. Dean, editor, *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*, pages 96–101, Stockholm, Sweden, 1999. Morgan Kaufmann, Los Altos.

[6] F. Baader, R. Küsters, and R. Molitor. Rewriting concepts using terminologies. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR-00)*, pages 297–308, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[7] F. Baader and R. Molitor. Building and structuring description logic knowledge bases using least common subsumers and concept analysis. In B. Ganter and G. Mineau, editors, *ICCS-00*, volume 1867 of *Lecture Notes in Artificial Intelligence*, pages 290–303. SV, 2000.

[8] F. Baader and P. Narendran. Unification of Concept Terms in Description Logics. In *Proceedings of ECAI-98*, pp. 331–335, John Wiley & Sons Ltd., 1998.

[9] A. Borgida and D. L. McGuinness. Asking Queries about Frames. In *Proceedings of KR'96*, pp. 340–349, Morgan Kaufmann Publishers, 1996.

[10] William W. Cohen, Alex Borgida, and Haym Hirsh. Computing least common subsumers in description logics. In William Swartout, editor, *Proc. of the 10th Nat. Conf. on Artificial Intelligence (AAAI-92)*, pages 754–760, San Jose, CA, 1992. AAAI Press/The MIT Press.

[11] R. Küsters and A. Borgida. What's in an attribute? Consequences for the least common subsumer. *JAIR*, 14:167–203, 2001.

[12] R. Küsters and R. Molitor. Approximating most specifc concepts in description logics with existential restrictions. In *Proc. of the 24th German Annual Conf. on Artificial Intelligence (KI'01)*, 2001. to appear.

[13] D.L. McGuinness. *Explaining Reasoning in Description Logics.* PhD thesis, Department of Computer Science, Rutgers University, October, 1996.

[14] Bernhard Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence Journal*, 43:235–249, 1990.

[15] U. Sattler. *Terminological knowledge representation systems in a process engineering application.* PhD thesis, LuFG Theoretical Computer Science, RWTH-Aachen, 1998

[16] A.-Y. Turhan and R. Molitor. Using lazy unfolding for the computation of least common subsumers. In *DL-2001*, 2001.